Profile-Driven Energy Reduction in Network-on-Chips

Feihui Li, Guangyu Chen, Mahmut Kandemir
Department of computer Science and Engineering
The Pennsylvania State University, USA
{feli,gchen,kandemir}@cse.psu.edu

Ibrahim Kolcu
Computation Department
University of Manchester, UK
ikolcu@umist.ac.uk

Abstract

Reducing energy consumption of a Network-on-Chip (NoC) is a critical design goal, especially for power-constrained embedded systems. In response, prior research has proposed several circuit/architectural level mechanisms to reduce NoC power consumption. This paper considers the problem from a different perspective and demonstrates that compiler analysis can be very helpful for enhancing the effectiveness of a hardware-based link power management mechanism by increasing the duration of communication links' idle periods. The proposed profile-based approach achieves its goal by maximizing the communication link reuse through compiler-directed, static message re-routing. That is, it clusters the required data communications into a small set of communication links at any given time, which increases the idle periods for the remaining communication links in the network. This helps hardware shut down more communication links and their corresponding buffers to reduce leakage power. The current experimental evaluation, with twelve data-intensive embedded applications, shows that the proposed profile-driven compiler approach reduces leakage energy by more than 35% (on average) as compared to a pure hardware-based link power management scheme.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—compilers; C.2.0 [*Computer-Communication Networks*]: General

General Terms Algorithms, Management, Experimentation

Keywords Network-on-Chip, power, compiler, routing

1. Introduction

The increasing complexity of communication patterns in embedded applications, which are parallelized over multiple processing units, creates difficulty for continued use of traditional bus-based on-chip communication techniques. Network-on-chip (NoC) architectures are rapidly replacing dedicated interconnects and buses in complex System-on-Chip (SoC) architectures. An NoC can connect and manage communications among a variety of design elements and Intellectual Property (IP) blocks required by complex SoCs. Prior research of NoCs mainly focused on circuit/architectural level techniques [2, 5, 31] and task mapping related issues [1, 13].

While, in principle, communication strategies similar to those currently used for large off-chip networks can apply at the chip level, increasing on-chip power consumption demands a power-aware design and an optimization process. Recent research shows that using voltage/frequency scaling on communication links [24] and shutting down the idle links [27] can significantly reduce NoC power consumption. 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.

Motivated by this preference, this paper proposes and evaluates 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. Note that a large fraction of embedded NoCs typically execute a single application and use static message routing for reasons such as energy efficiency and buffer space minimization [13]. This paper proposes a profile-driven static message routing scheme that maximizes link reuse between different execution states of a given application. Our approach makes use of a novel data structure called the communication graph (which captures different network states during application execution) and a new abstraction: the "link signature" (which captures link utilization caused by messages in a given network state). We implement our approach and test it using twelve data-intensive embedded applications. Analysis of the test results show that the proposed profile-driven compiler approach reduces leakage energy by more than 35% on average, as compared to a pure hardware-based link power management scheme.

The remainder of this paper is organized as follows. Section 2 introduces our on-chip mesh architecture and hardware support for the compiler-directed message routing. Section 3 explains how link signatures and the communication graph derive from an automated compiler analysis and describes the link reuse optimization approach. Section 4 presents the experimental results from our implementation. Sections 5 and 6 describe related work and provide concluding remarks, respectively.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PLDI'07 June 11–13, 2007, San Diego, California, USA.
Copyright © 2007 ACM 978-1-59593-633-2/07/0006...$5.00

2. Architectural Model

2.1 Network Abstraction

Our research focuses on a mesh-based NoC architecture, whose abstract view appears in Figure 1. Each node in the mesh has a simple, single-issued, embedded core with a small amount of local memory (32KB in our experiments). Each node is connected to its local switch through a network interface (NI) and thus communicates with other nodes through switches and communication links. Interprocessor communication is done via message passing.

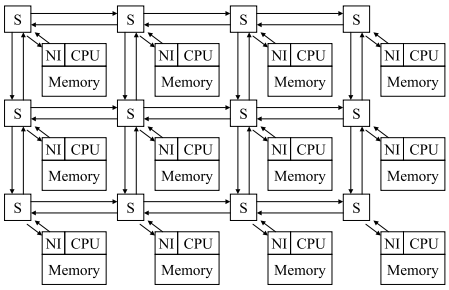


Figure 1. Mesh-based NoC architecture (S: switch; NI: network interface).

The internal structure of a switch is shown in Figure 2. Each switch has an interface with its local processing unit. It also has four input/output ports that interface with four neighboring switches. Input/output buffers in a switch store the packets in transmission. A cross-bar decides to which output port a packet is to be sent. The default routing algorithm used by the switches is static X-Y routing [9], which passes each packet first exclusively in the X-dimension and then completely in the Y-dimension until reaching the destination node.

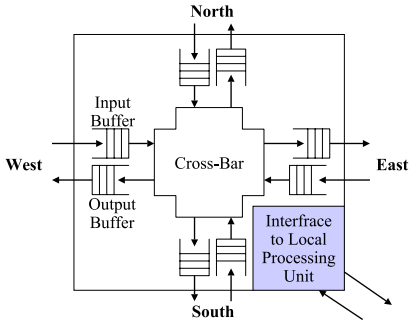


Figure 2. Switch structure.

Based on prior research of power-aware networks [15, 27], we employ a hardware-controlled link shutdown scheme. 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.

2.2 Hardware Support for Compiler-Directed Message Routing

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 [29]). Please note that the selection of the communication library to use is orthogonal to the focus of this paper.

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 3. When a switch receives such a packet, it forwards the packet according to the X-Y routing algorithm.

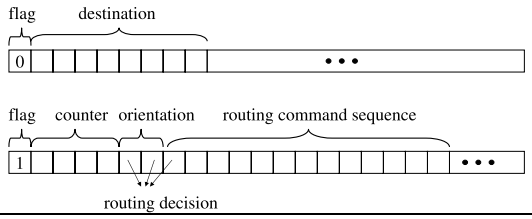


Figure 3. Fields in the header of a packet (Top: default X-Y routing; Bottom: compiler-directed routing).

Table 1. 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

On the other hand, the header of a packet that employs the compiler-directed routing contains three fields (see the lower part of Figure 3): 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. 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 1 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.

3. Optimizing Link Reuse

A high level view of our compiler-based approach appears in Figure 4. 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 (elaboration of the concepts of link signature and communication graph appears

later in this section). 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.

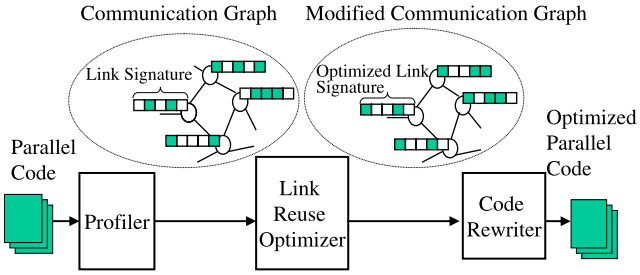


Figure 4. Compiler-directed link reuse optimization scheme. Each vertex of the communication graph captures a network state.

3.1 Network State and Link Signature

Assume that a parallel program to be executed 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:

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

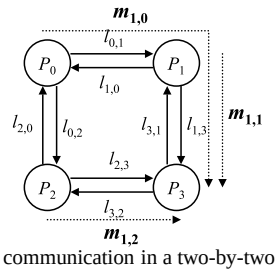
$S_0 = \phi$ represents a state in which no message is in transmission.

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 5 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 5(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 5(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).



(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).

Figure 5. A link signature calculation example.

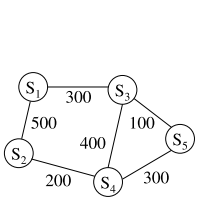
3.2 Communication Graph

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:

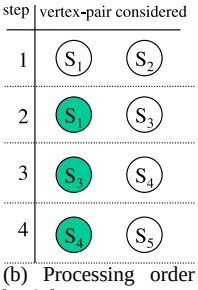
- A new message is sent by communication command, $\mathcal{M}_{k,p}$. In this case, $S_j = S_i \cup \{\mathcal{M}_{k,p}\}$.
- A message sent by communication command $\mathcal{M}_{k,p}$ arrives at its destination node. In this case, $S_j = S_i - \{\mathcal{M}_{k,p}\}$.

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.

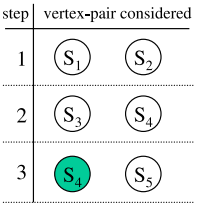
We use *profiling* to build the CG of a parallel program. Specifically, we instrument the target program to notify a profiler each time a node sends a message or when a message arrives its destination node. The profiler keeps track of the current network state, S_i. When the profiler receives a notification 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 6(a) illustrates an example communication graph.



(a) A communication graph.



(b) Processing order by Scheme I.



(c) Processing order by Scheme II.

Figure 6. Two different approaches to traverse a CG (a shaded vertex at step k indicates that the corresponding link signature is not modified at this step).

3.3 Maximizing Link Reuse

Based on the previous definitions, we can re-express the problem of increasing link reuse 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, which will maximize communication link reuse.

3.3.1 Traversing a 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 with the largest weight and performs the signature re-assignment to the associated vertices. After that, this approach considers the edge with the next largest weight among the edges that are incident on the 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 6(a). The pairs of vertices considered by Scheme I and Scheme II at each step (for signature re-assignment) appear in Figure 6(b) and Figure 6(c), respectively. Figure 7(a) and Figure 7(b) provide the pseudo-codes for the compiler algorithms that implement Scheme I and Scheme II, respectively.

3.3.2 Optimizing Link Reuse Between Two Adjacent Network States

1) Routing Flexibility. Re-routing (the messages sent by) the 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 3.1 that the defined 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 as the replacement of the current LUV for 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, $\mathcal{M}_{i,p}$.

2) 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 8. 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.

```
Input:
  A communication graph CG(V, E, W);
Output:
  ū_{i,p} for each 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 = φ; R = φ; C = φ;
while(P ≠ V) {
  if(C = φ)
    C = {(S_x, S_y)} if W_{x,y} is maximum;
  select (S_i, S_j) ∈ C with maximum W_{i,j};
  call reroute(S_i, S_j, R);
  P = P ∪ {S_i, S_j};
    // states S_i and S_j have been processed.
  R = R ∪ S_i ∪ S_j;
    // the LUVs of the communication commands
    // in S_i and S_j have been determined.
  C' = {(S_a, S_b)|S_a ∈ P ∧ S_b ∈ (V − P)};
  C = (C − {(S_i, S_j)}) ∪ C';
}
```

(a) Scheme I.

```
Input:
  A communication graph CG(V, E, W);
Output:
  ū_{i,p} for each 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 = φ; R = φ; C = E;
while(P ≠ V) {
  select a (S_i, S_j) ∈ C if W_{i,j} is maximum;
  call reroute(S_i, S_j, R);
  P = P ∪ {S_i, S_j};
    // states S_i and S_j have been processed.
  R = R ∪ S_i ∪ S_j;
    // the LUVs of the communication commands
    // in S_i and S_j have been determined.
  C = C − {(S_i, S_j)};
}
```

(b) Scheme II.

Figure 7. Pseudo codes for our two CG traversing schemes (Scheme I and Scheme II).

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 influences the latency of transmitting messages. The link with the highest load corresponds

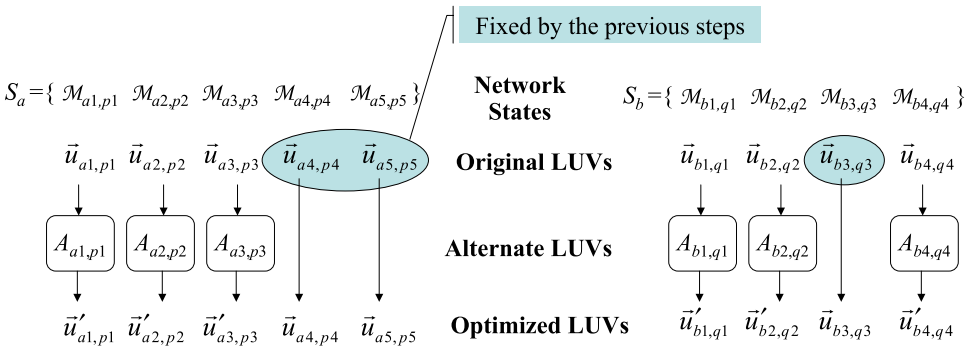


Figure 8. Link reuse optimization between two network states, S_a and S_b.

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.

3) 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 9.

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 rerouted 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

```
Input:
  S_a, S_b — two network states;
  R        — the set of communication commands whose LUVs
             have been determined
Output:
  u_{i,p}  — LUV for each M_{i,p} ∈ ((S_a ∪ S_b) − R).

// determine the LUV for each communication command
// in states S_a and S_b if it has not been determined yet.
procedure reroute(S_a, S_b, R) {
  for each M_{i,p} ∈ (S_a ∪ S_b − R) {
    calculate u_{i,p}, the LUV of M_{i,p}, based on X-Y routing;
    calculate A_{i,p}, the ALUV of M_{i,p};
    calculate s_a, the link signature of state S_a;
    calculate s_b, the link signature of state S_b;
    num_links = |θ(s_a) ∪ θ(s_b)|;
    if(θ(s_a) ⊆ θ(s_b) ∨ θ(s_b) ⊆ θ(s_a))
      return;
    sort all M_{i,p} ∈ (S_a ∪ S_b − R) by the routing flexibility |A_{i,p}|
    for each M_{i,p} ∈ (S_a ∪ S_b − R){
      for each v ∈ A_{i,p} {
        if M_{i,p} ∈ S_a ∧ M_{i,p} ∈ S_b {
          calculate s_{a_new} by using v as LUV of M_{i,p}
          calculate s_{b_new} by using v as LUV of M_{i,p}
          if(max(s_{a_new}) > max(s_a)) continue;
          if(max(s_{b_new}) > max(s_b)) continue;
          if(|θ(s_{a_new}) ∪ θ(s_{b_new})| ≥ num_links) continue;
          if(|θ(s_{a_new}) ∪ θ(s_{b_new})| = num_links∧
              |θ(s_{a_new}) ∩ θ(s_{b_new})| ≤ |θ(s_a) ∩ θ(s_b)|)
            continue;
          replace u_{i,p} with v;
          s_a = s_{a_new}; s_b = s_{b_new};
          num_links = |θ(s_a) ∪ θ(s_b)|;
        } else {
          if(M_{i,p} ∈ S_a) { x = a; y = b; }
          else { x = b; y = a; }
          calculate s_{x_new} by using v as LUV of M_{i,p}
          if max(s_{x_new}) > max(s_x) continue;
          if(|θ(s_{x_new}) ∪ θ(s_y)| > num_links) continue;
          if(|θ(s_{x_new}) ∪ θ(s_y)| = num_links∧
              |θ(s_{x_new}) ∩ θ(s_y)| ≤ |θ(s_a) ∩ θ(s_b)|)
            continue;
          replace LUV of M_{i,p}, i.e., u_{i,p}, with v;
          s_x = s_{x_new};
          num_links = |θ(s_x) ∪ θ(s_y)|;
        }
      }
    }
  }
}
function max(v) {
  return the value of the largest entry of v;
}
```

Figure 9. Communication link reuse optimization heuristic.

computed for state S_a. Subsequently, using this new signature, 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.

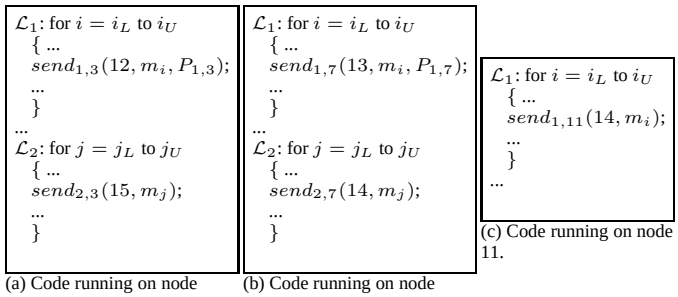
3.4 Example

This section provides an example 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 10(a) and Figure 10(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 10(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 10(a) and Figure 10(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 10(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 10(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 10(b). Thus, we do not show the result of step III in Figure 10. 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 10(f) and Figure 10(g), respectively. Each step reduces the total number of links used in the two network states (i.e., improves link reuse). Figure 10(g) gives the final routings for all the communication commands in state S_a. The modified LUVs and LSs returned by this method are given in Figure 10(h). Clearly, the total number of links used in states S_a and S_b decreases from 16 to 12.

3.5 Code Rewriter

This component of our approach (see Figure 4) is responsible for providing a version of the message send operation, which incorporates the compiler-determined routing information. The code fragments shown in Figure 11 correspond to the example in Figure 10. 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 3 and Table 1. Therefore, all the messages sent by operation $send_{1,3}$ have the message header: 1011011000111000000; 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.



(a) Code running on node 3.

(b) Code running on node 7.

(c) Code running on node 11.

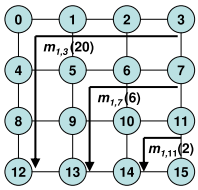
Figure 11. Code rewriting for the example in Figure 10.

3.6 Handling Deadlocks

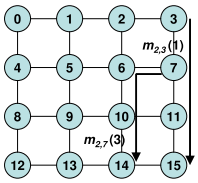
An important issue that this scheme must address is how to handle possible deadlocks, as the re-routings change the behavior of the default X-Y routing scheme, which is a deadlock-free routing algorithm [6]. Dally and Seitz [6] proved that an acyclic channel dependency graph is the necessary and sufficient condition for avoiding deadlocks. Thus, adding a simple deadlock handling procedure (Figure 12) breaks the possible cycles within the channel dependency graph by changing some messages' routings. This procedure applies after using the rerouting algorithm in Figure 9. Checking the routing paths within each state of the two network states in question identifies a cyclic channel waiting. If none exists, the procedure simply returns, indicating no possibility of deadlock. On the other hand, if there exists cyclic channel waiting (possible deadlocks), the deadlock handling procedure reviews all the messages causing cyclic channel waiting. For each message, the algorithm checks for an alternate path breaking the cycle in the channel dependency graph and, at the same time, without increasing the number of links used by the two network states. If such a path exists, it replaces the original path; if not, the algorithm simply returns.[1]

An important observation at this point is that, while the previously explained deadlock handling procedure helps reduce the

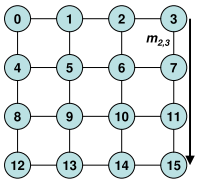
[1] A better approach would be to search for a path that eliminates the potential deadlock but does not necessarily use the same number of links. However, this would make the algorithm much more complex.



(a) Network state S_a.



(b) Network state S_b.

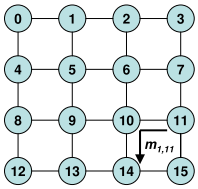


(c) Re-routing step I (state S_b).

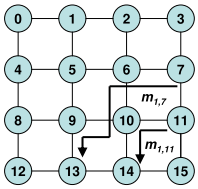
Links:	$l_{3,2}$	$l_{2,1}$	$l_{1,0}$	$l_{0,4}$	$l_{4,8}$	$l_{8,12}$	$l_{7,6}$	$l_{6,5}$	$l_{5,9}$	$l_{9,13}$	$l_{11,10}$	$l_{10,14}$	$l_{3,7}$	$l_{7,11}$	$l_{11,15}$	$l_{6,10}$	...
$\vec{u}_{1,3}$:	(20	20	20	20	20	20	0	0	0	0	0	0	0	0	0	0	...)
$\vec{u}_{1,7}$:	(0	0	0	0	0	0	20	20	20	20	0	0	0	0	0	0	...)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

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

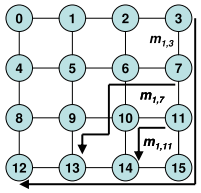
(d) The link utilization vectors and link signatures using default X-Y routing. $\vec{s}_a$ and $\vec{s}_b$ are the signatures for state S_a and S_b, respectively. The total number of links used by the two states is 16. Note that the LUV entries not shown above explicitly are zero.



(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	...)

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

(h) The link utilization vectors and link signatures after re-routing. $\vec{s'}_a$ and $\vec{s'}_b$ are the new signatures determined by our approach for state S_a and state S_b, respectively. The total number of links being used by the two states is 12. Note that the LUV entries not shown above explicitly are zero.

Figure 10. An example that illustrates how our approach works. (a) and (g) represent the default routings and compiler-determined routings of network state S_a, respectively. (b) represents the default routings of network state S_b (the compiler does not change the routings of S_b in this example).

probability of experiencing a deadlock at runtime, it cannot completely eliminate deadlocks. This is because the rerouting algorithm is profile driven, and a new input to the application can change the execution behavior. As a result, a runtime based, deadlock handling approach is needed. To ensure fully deadlock-free execution, we use the dynamic, hardware-supported deadlock avoidance rule employed by the Alpha 21364 network architecture [18]. This rule, based on a theory proposed by Duato [8], states that a cyclic channel dependency graph does not lead to deadlocks if packets can drain via a deadlock-free path. By using virtual channels (separate buffer queues), logically distinct networks, which include an adaptive network and a deadlock-free network, are constructed. In this approach, for performance reasons, most bandwidths are assigned to the adaptive network (formed by adaptive virtual channels with no routing limitation), while the deadlock-free network (formed by deadlock-free virtual channels with routing restriction) used to drain deadlocked packets occupies limited bandwidth. Since a deadlock handling procedure already runs after rerouting (as explained above), a limited deadlock-free bandwidth is sufficient for draining the infrequent deadlocks. The important point is that this dynamic approach does not incur extra cycle costs or energy consumption as long as no deadlocks occur at runtime. When deadlocks occur, however, draining the stuck messages results in both extra latency and power consumption (due to leakage).

4. Experiments

4.1 Simulation Environment and Benchmarks

To conduct the experiments, we implemented a flit-level on-chip interconnection network simulator. The network, parametrized similar to that in [5, 7], 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 [27]. The time-out 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

Input:
S_a, S_b — two network states;
R — the set of communication commands whose LUVs have been determined
$\vec{u}_{i,p}$ — LUV for each $\mathcal{M}_{i,p} \in (S_a \cup S_b)$ decided by the rerouting algorithm in Figure 9.
Output:
$\vec{u}_{i,p}$ — LUV for each $\mathcal{M}_{i,p} \in ((S_a \cup S_b) - R)$.

```
procedure handle_deadlock() {
  if ( !HasCyclicWait(S_a) ∧ !HasCyclicWait(S_b) ) {
    return;
  } else {
    calculate s⃗_a, the link signature of state S_a;
    calculate s⃗_b, the link signature of state S_b;
    num_links = |θ(s⃗_a) ∪ θ(s⃗_b)|;
    for each M_{i,p} ∈ D (D: minimal subset of (S_a ∪ S_b) − R
      causing channel cyclic waiting) {
      calculate A_{i,p}, the ALUV of M_{i,p};
      for each v⃗ ∈ A_{i,p} {
        calculate s⃗_{a_new} by using v⃗ as LUV of M_{i,p}
        calculate s⃗_{b_new} by using v⃗ as LUV of M_{i,p}
        if(|θ(s⃗_{a_new}) ∪ θ(s⃗_{b_new})| = num_links∧
          !HasCyclicWait(S_a) ∧ !HasCyclicWait(S_b) )
          replace u⃗_{i,p} with v⃗;
        }
      }
    }
  }
}
```

Figure 12. Reducing potential deadlocks.

as 1μsec, and the energy overhead of this switching is assumed to be 140μJ, based on prior research [5, 27]. This study's simulator uses the on-chip, interconnection network power model proposed by Chen and Peh [5]. When a link is turned off, it consumes zero leakage energy. The network energy model employed is not a major contribution of this paper and requires no further elaboration. 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 [11]. 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 [30]. 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 [29]. Having determined the code fragment that will be executed by each processor, invoking the approach discussed in this paper follows. This approach determines the link signatures, builds the communication graph, and performs message re-routing, as explained earlier. Both communication graph traversal schemes (Scheme I and Scheme II), discussed in Section 3.3.1, 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 [27, 15, 5]. In this implementation, parameters are selected such that the energy savings achieved by

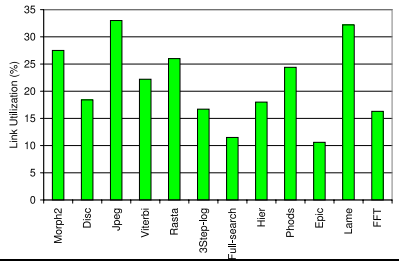


Figure 13. Link utilization.

link shutdown are maximized without unnecessarily hurting network latency. In the rest of this paper, 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, as discussed earlier in Section 3.3.1. 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. A common characteristic of these benchmarks is their array/loop-intensive embedded application nature. The first five benchmarks are collected from different sources, the next four from [33] and the last three codes are the only array-based codes in the MediaBench [16] and MiBench [17] suites. The second column shows a brief description of each benchmark. 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.

The 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, respectively. That is, results are normalized with respect to the corresponding results of the base scheme that implements the hardware-based link shutdown. We want to emphasize that 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 offline activities, these increases are within acceptable range. In none of the experiments we conducted (even the ones with the different input sets than those used for profiling), we observed any deadlock. That is, our static deadlock elimination technique was very effective in practice.

4.2 Results

Figure 13 presents the average link utilization (the fraction of the cycles in which the links are used for transferring packets). The

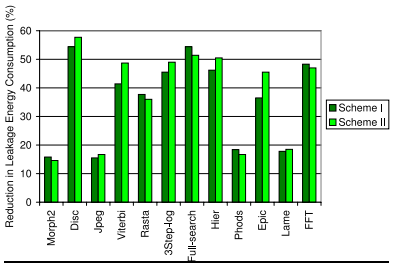
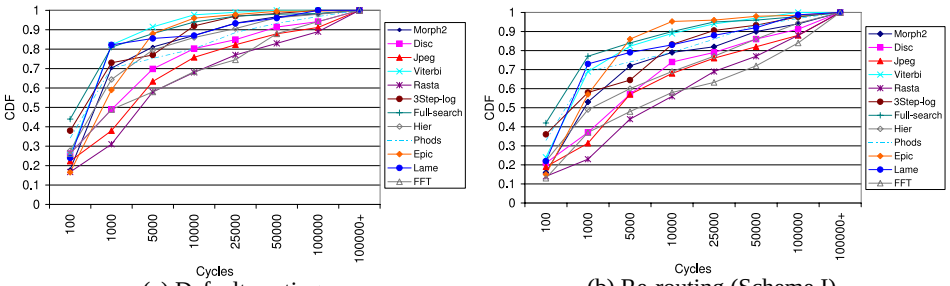


Figure 14. Percentage reductions in leakage energy consumption.



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

Figure 15. CDF for link idle periods.

Table 2. 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%)

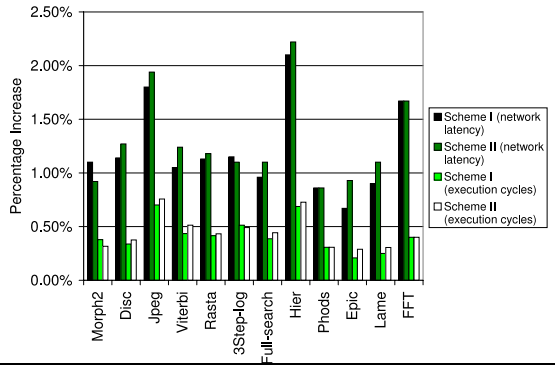


Figure 16. Percentage increases in network cycles and overall execution time.

average link utilization for the benchmarks 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, as explained earlier, the 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 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 14, 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 discussed in Section 3.3.1: Scheme I and Scheme II. From these results, the average leakage energy savings, when applying the compiler-directed message re-routing,

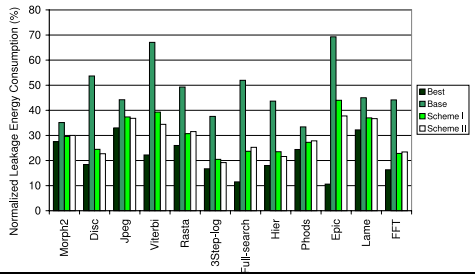


Figure 17. Leakage energy consumptions.

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 15(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 from these plots 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 16 (the network latency increases due to the base scheme itself appear in the last column of Table 2). From these results, 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 16 is that the average increase in overall execution time is less than 0.5% for both Scheme I and Scheme II.

Figure 17 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

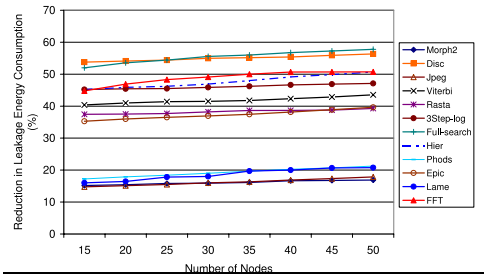


Figure 18. Sensitivity to the number of nodes (Scheme I). The results with Scheme II are similar.

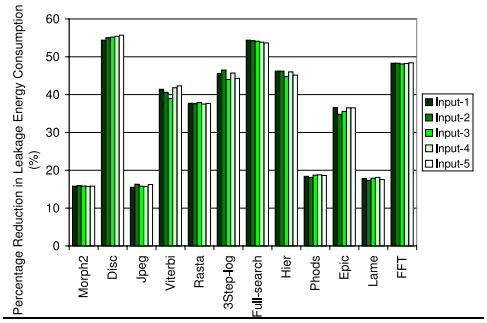


Figure 19. Sensitivity to the input size (Scheme I). The results with Scheme II are similar.

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.

The graph in Figure 18 plots the leakage energy savings for Scheme I with different numbers of nodes. The default mesh used so far in the experiments has 25 (=5×5) nodes. All the curves are normalized with respect to the base case with a corresponding number of nodes. The results from Scheme II are very similar to those presented in Figure 18, so they are omitted. The leakage energy savings obtained from different mesh sizes are similar, mainly because these represent normalized results with respect to the base case, which already adopts a leakage saving scheme (and thus takes advantage of the additional idle links introduced by a larger mesh). Still, as the number of nodes increases, slight increases in savings occur.

The final measurement is the input sets' effect on energy savings. Such an analysis is important because the proposed approach is profile-based and a different input set can generate different network states than those obtained by the input set used in profiling. Figure 19 presents the results from Scheme I. In this graph, Input-1 through Input-5 correspond to the results of different input sets (Input-1 being the default used in profiling). These results show that 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.

5. Discussion of Related Work

In recent years, several efforts have attempted to minimize energy consumption of the NoC based systems and chip-to-chip networks. For example, Simunic and Boyd [25] proposed a network-centric power management scheme for NoCs. Their experimental results demonstrate that this technique can predict future workloads more accurately than node-centric power management schemes. Soteriou and Peh [26, 27] explored the design space for communication link turn-on/off, based on a dynamic power management technique. They proposed a design methodology for power-aware interconnection networks. Worm et al [32] proposed an adaptive low-power transmission scheme for on-chip networks. Their goal was to minimize the energy required for reliable communications, while satisfying a QoS constraint by dynamically varying the voltages of the links. Kim et al [15] designed a link shutdown scheme that minimized the number of active links while maintaining the connectivity of the network. They made use of an adaptive routing algorithm, and presented a detailed comparison of the proposed scheme with voltage scaling. Shang et al [23] proposed applying dynamic voltage scaling to communication links. They used a history-based policy to lower the voltages of the links with low utilization. In a sense, the present research is complementary to these previous efforts. Since this new approach increases link idle periods, the expectation is that any link shutdown or voltage scaling based hardware mechanism is more effective when used in conjunction with the proposed method.

Another group of related work is power modeling for interconnection networks. Eisley and Peh [10] proposed LUNA, a high-level power analysis framework for on-chip networks. Wang et al [31] presented an architectural-level power-performance simulator for interconnection networks. Using this simulator, their paper evaluated different network architectures and the impact of different communication patterns on energy consumption. Patel et al [21] focused on the power-constrained design of interconnection networks and proposed power models for routers and links. Raghunathan et al [22] presented a survey of energy-efficient on-chip communication techniques that function across the different levels: circuit-level, architecture-level, system-level, and network-level. In contrast to these studies, the goal in this research is to explore the role of a compiler in reducing the NoC energy consumption.

Prior compiler work [28, 19] for chip multi-processors focused mainly on improving performance. Jalabert et al [14] designed a tool called xpipes-Compiler, for instantiating application-specific NoCs. Shin and Kim [24] use different algorithms to explore design space for NoC systems. Hu and Marculescu [13] proposed an algorithm that maps a given set of IP blocks onto a regular NoC structure and constructed a routing function that minimized communication. The focus of these studies is to reduce energy consumption via task mapping. Our approach is different from these others in that it focuses on reducing energy consumption through compiler-directed communication link reuse.

Chen et al [3] presented a compiler method that performed energy efficient channel allocation under performance bounds. Compared to that static analysis, the proposed scheme uses profiling information to identify optimization opportunities. In addition, the approach can assign different routing paths to different message sending operations, while Chen et al assigns a single fixed path for each source-destination node pair. In fact, their approach reduces the leakage energy consumption by about 20%, on average,

over a hardware scheme that already performs link shutdown. As given in Section 4, the normalized leakage energy consumption of our approach and the hardware link shutdown method are around 30% and 48%, respectively. That is, the proposed scheme achieves a leakage energy reduction of about 37% over and above the hardware link shutdown scheme, demonstrating the importance of selecting routes based on individual send operations, rather than fixing it based on source-destination pairs (i.e., 37% saving versus 20% saving). Finally, Chen et al [4] presents a compiler directed voltage scaling model which can be combined with the link shutdown approach proposed in this paper to reduce NoC energy consumption even further. Other recent software-based techniques for NoCs include [12, 20].

6. Conclusions

The main contribution of this research is a profile-driven compiler scheme that increases energy benefits obtained from a hardware-based communication link shutdown mechanism. The proposed compiler-based approach achieves its goal by determining the routes of the communication messages at compile time in such a fashion that link reuse between messages is maximized without significantly affecting network performance. In other words, the approach limits link usage, at a given time, to a small set of links, and the remaining links shut down to save power. The experimental evaluation using twelve embedded applications shows that the proposed approach is quite successful in practice.

Acknowledgments

We would like to thank Seth C. Goldstein and anonymous reviewers for their valuable comments. This work is supported in part by NSF Career Award 0093082.

References

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

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

[3] G. Chen, F. Li, and M. Kandemir. Compiler-directed channel allocation for saving power in on-chip networks. In *Proc. 33rd Annual Symposium on Principles of Programming Languages*, 2006.

[4] G. Chen, F. Li, M. Kandemir, and M. J. Irwin. Reducing noc energy consumption through compiler-directed channel voltage scaling. In *Proc. the 2006 ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 193–203, New York, NY, USA, 2006. ACM Press.

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

[6] W. J. Dally and C. L. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Trans. Comput.*, 36(5):547–553, 1987.

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

[8] J. Duato. A new theory of deadlock-free adaptive routing in wormhole networks. *IEEE Trans. Parallel and Distributed Systems*, 4(12):1320–1331, 1993.

[9] J. B. Duato, S. Yalamanchili, and L. Ni. *Interconnection Networks*. Morgan Kaufmann Publishers, 2002.

[10] 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*, Sept. 2004.

[11] P. S. M. et al. Simics: A full system simulation platform. *Computer*, 35(2):50–58, 2002.

[12] A. Hansson, K. Goossens, and A. Rdulescu. A unified approach to constrained mapping and routing on network-on-chip architectures. In *Proc. International Conference on Hardware/Software Co-Design and System Synthesis*, 2005.

[13] 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), Apr. 2005.

[14] A. Jalabert, S. Murali, L. Benini, and G. D. Michieli. XpipesCompiler: A tool for instantiating application specific Networks-on-Chip. In *Proc. the Conf. on Design, Automation and Test in Europe*, 2004.

[15] 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*, Aug. 2003.

[16] http://cares.icsl.ucla.edu/MediaBench/.

[17] http://www.eecs.umich.edu/mibench/.

[18] S. S. Mukherjee, P. Bannon, S. Lang, A. Spink, and D. Webb. The alpha 21364 network architecture. *IEEE Micro*, 22(1), Jan. 2002.

[19] R. Nagarajan, D. Burger, K. S. McKinley, C. Lin, S. W. Keckler, and S. K. Kushwaha. Static placement, dynamic issue (SPDI) scheduling for EDGE architectures. In *Proc. International Conference on Parallel Architectures and Compilation Techniques*, Oct. 2004.

[20] U. Ogras, J. Hu, and R. Marculescu. Key research problem in NoC design: A holistic perspective. In *Proc. International Conference on Hardware/Software Co-Design and System Synthesis*, 2005.

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

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

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

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

[25] T. Simunic and S. Boyd. Managing power consumption in networks on chip. In *Proc. the Conf. on Design, Automation and Test in Europe*, 2002.

[26] V. Soteriou and L.-S. Peh. Dynamic power management for power optimization of interconnection networks using on/off links. In *Proc. Symposium on High Performance Interconnects*, 2003.

[27] V. Soteriou and L.-S. Peh. Design space exploration of power-aware on/off interconnection networks. In *Proc. the 22nd Int. Conf. on Computer Design*, Oct. 2004.

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

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

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

[31] 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*, Nov. 2002.

[32] F. Worm, P. Ienne, P. Thiran, and G. D. Micheli. An adaptive low power transmission scheme for on-chip networks. In *Proc. International System Synthesis Symposium*, 2002.

[33] N. D. Zervas, K. Masselos, and C. Goutis. Code transformations for embedded multimedia applications: impact on power and performance. In *Proc. ISCA Power-Driven Microarchitecture Workshop*, 1998.