Exploiting Last Idle Periods of Links for Network Power Management*

F. Li, G. Chen, M. Kandemir
CSE Department
Pennsylvania State University
University Park, PA 16802, USA
{feli,gchen,kandemir}@cse.psu.edu

M. Karakoy
Department of Computing
Imperial College
London, SW7 2AZ, UK
m.karakoy@ic.ac.uk

ABSTRACT

Network power optimization is becoming increasingly important as the sizes of the data manipulated by parallel applications and the complexity of inter-processor data communications are continuously increasing. Several hardware-based schemes have been proposed in the past for reducing network power consumption, either by turning off unused communication links or by lowering voltage/frequency in links with low usage. While the prior research shows that these schemes can be effective in certain cases, they share the common drawback of not being able to predict the link active and idle times very accurately. This paper, instead, proposes a compiler-based scheme that determines the last use of communication links at each loop nest and inserts explicit link turn-off calls in the application source. Specifically, for each loop nest, the compiler inserts a turn-off call per communication link. Each turned-off link is reactivated upon the next access to it. We automated this approach within a parallelizing compiler and applied it to eight array-intensive embedded applications.

Categories and Subject Descriptors

D.3.4 [**Software**]: Programming Language—*Processors,Optimization*

General Terms

Languages

Keywords

interconnection network, energy optimization, compiler

1. INTRODUCTION

Increasing energy consumption of parallel architectures makes network power consumption an important target for optimization. While network performance has received a lot of attention in the past from both hardware and software communities, network power optimization is a relatively new topic [1, 2, 8, 6]. To date, several hardware-based schemes have been proposed for reducing network power consumption, by either turning off unused communication links or lowering voltage/frequency in links with low usage [12, 4, 7]. While the prior research shows that the hardware-based network power optimization schemes can be very effective in certain cases [10], they share a common drawback that they cannot extract the high level network usage patterns (e.g., the order and timing of link usages). As a result, they may be inefficient in certain cases by reacting too late to the variations in network usage patterns. The main goal of this paper is to explore whether a software-directed approach to network power management is possible.

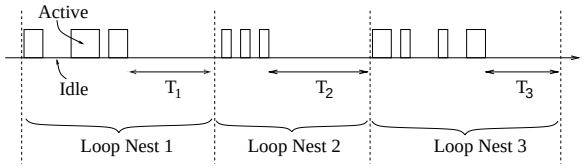

Figure 1: An example link access pattern for a program with three loop nests.

Figure 1 depicts the access pattern for a given communication link in a sample program fragment with three separate loop nests. In this figure, the time is assumed to progress from left to right, and each active period (during which the link is active in communicating a message) of the link is specified using a pulse whose width represents the length of the active period. The remaining periods are idle periods (during which the link is not used for communication). Our focus is on idle periods T_1, T_2, and T_3. A common characteristic of these three idle periods is that they are the *last idle periods* (*times*) in each nest, and represent the time frame between the last use of the link and the end of the loop nest.

Figure 2 shows the communication link energy distribution between active periods and idle periods. One can observe from this graph that the most of link energy consumption is spent during idle periods (78.4% on an average). This result can be explained as follows. In message-passing architectures, the application code is generally parallelized in such a fashion that the interprocessor communication is minimized as much as possible. To achieve this, communication optimizations such as message vectorization, message aggregation, and message coalescing [5] are used. As a consequence of this, most of the communication links are idle at a given time, and are responsible from a significant portion of the link energy consumption.

Figure 3 shows the the energy contribution of the last idle periods. In Figure 1, for example, this contribution amounts to $\{(E_1 + E_2 + E_3)/E_{total-idle}\} \times 100$, where E_1, E_2, and E_3 correspond to the energy consumptions in idle periods T_1, T_2, and T_3, respectively. $E_{total-idle}$, on the other hand, captures the total idle time energy when considering all three nests in the program (including the energies consumed in the last idle times as well). An interesting observation from Figure 3 is that a very significant portion of the energy spent in idle times are spent in the last idle times, specifically, 60.2% when averaged over the eight benchmarks. This indicates that a scheme that can exploit these last idle periods can be

*This work is supported in part by NSF Career Award 0093082 and a grant from GSRC.

EMSOFT'05, September 19–22, 2005, Jersey City, New Jersey, USA.

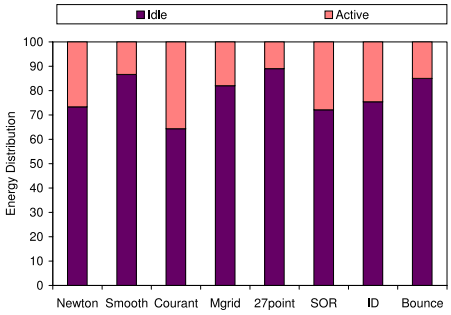

Figure 2: Distribution of link active and idle energies.

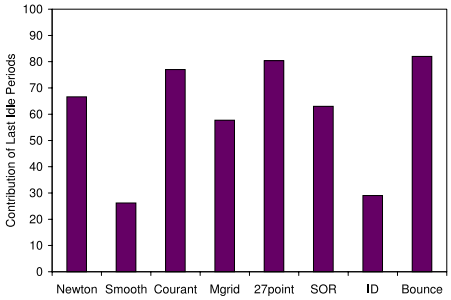

Figure 3: Energy contribution of the last idle periods (times), when accumulated over all the loop nests in the application.

very useful in practice. The next section presents such a scheme embedded within a parallelizing compiler.

Based on this observation, we propose a *compiler-directed* communication link shut-down (turn-off) strategy for reducing network energy. The proposed strategy takes as input a parallelized application code (using explicit message-passing directives such as those supported by MPI [3]). It first analyzes the interprocessor communication pattern of the input code and identifies the link access pattern, i.e., the order and timing of communication link usage. It then inserts explicit link turn-off calls in the application source. Specifically, for each loop nest in the application code, the compiler inserts a call that shuts down the links after their last uses. Each turned-off link is reactivated upon the next access to it. Our approach targets at small networks that are used by a single application at a time.

The next section explains the network abstraction used by our approach. It also discusses the architectural support needed by our compiler based scheme. Section 3 presents our compiler-based approach for exploiting this information by turning off the communication links that completed their last uses (communications) in a given loop nest. Finally, Section 4 summarizes our major observations and gives pointers for future research directions.

2. NETWORK ABSTRACTION AND HARDWARE SUPPORT

In this section, we discuss the network abstraction our compiler-directed approach uses and the architectural support needed. We focus on an $M \times N$ (M rows, N columns) mesh architecture[1] as depicted in Figure 4. Each node in the mesh consists of a processor, a memory module, and a switch.[2] The node at the i^{th} ($i = 0, 1, ..., M-1$) row and j^{th} ($j = 0, 1, ..., N-1$) column is labeled with an integer ID: $i \times N + j$.

Figure 5 gives the structure of a switch. Each switch has five in-coming ports (In-0 through In-4) and five out-going ports (Out-0 through Out-4). The ports In-0 and Out-0 are connected to the local processor (the processor in the same node as the switch). The remaining four in-coming ports and four out-going ports are connected to the switches in the neighboring nodes by a set of wires. A switch also provides a power control API that allows the local processor to turn on/off each out-going port of this switch. As will be discussed later, when an out-port is turned on/off, its corresponding in-port (in the switch of a neighbor node) is also turned on/off.

In this paper, we define $link(i, j)$ as the directed physical connection from a node (N_i) to one of its neighbors (N_j). We refer to nodes N_i and N_j as the sender and receiver of $link(i, j)$, respectively. In a mesh, each pair of adjacent nodes, N_i and N_j, are connected by a pair of links, namely, $link(i, j)$ and $link(j, i)$. Each link consists of a pair of ports (an out-going port of the sender switch and an in-coming port of the receiver switch) and the wires that connect these two ports.

Figure 6 shows the structure of a communication link from node N_i and N_j. A message to be transferred by a link is first stored in the buffer of the out-going port of this link. The power control logic in the out-going port checks the one-bit "CTRL" in the header of this message. CTRL=0 indicates that the message in the buffer is a *data message* that carries data that will be used by an application process. This message is forwarded without any modification (i.e., as it is) from one link to another until it arrives at its destination, and the contents of this message does not affect the power state of any communication link on its path. CTRL=1, on the other hand, indicates that the message in the buffer is a *power control message*. The body of this message contains a *vector*, each bit of which controls the power state of a link in the path from the source node to the destination node. This message is discarded by the switch on the destination node, and thus it is never received by any application process. When the power control logic detects that the message in the buffer is a power control message, it signals the buffer to shift the entire message body (not including the header) by one bit. The first bit of the message body, which is shifted out of the buffer, is stored in the one-bit *power state register* of the power control logic. The shifted message is then forwarded to the in-coming port, which is on the other end of this link. After that, the power control logic sets the power state of the Tx unit based on the value of the power state register. Specifically, if the value of the power state register is one, the power control logic turns off the Tx unit immediately; otherwise, the Tx unit remains in the active state. The link state monitor of the in-coming port monitors the state of the Tx unit. It turns on (off) the Rx unit when it detects that the corresponding Tx unit has been turned on (off). In addition to controlling the power state of a link using power control messages, a switch also provides a programming interface for the local processor to turn on/off each out-going port of this switch without sending any message.

A parallel program consists a set of parallel processes running on different nodes of the mesh. A process sends messages to another process through a logical connection (or connection for short). A logical connection consists of multiple links if the sender and receiver processes are running on two nodes that are not adjacent to each other. We use $C(i, j)$ to denote the set of links in the connection from the source node N_i to the destination node N_j. Since a connection can be unambiguously identified by the set of links used in this connection, we also use $C(i, j)$ to denote the connection from N_i to N_j. Note that, using the power control messages, we can control the power state of each link in any connection.

3. COMPILER SUPPORT

In this section, we discuss the details of our compiler algorithm that exploits the last idle periods of communication links. We focus on the array-based, loop-intensive embedded programs parallelized over a mesh architecture. Such a program consists of a set of parallel processes running on different nodes in a mesh. Each process consists a set of loop nests. These processes communicate with each other through inter-node connections. We assume that all the communication links in the mesh are initially turned off. A link used by a process is turned on automatically upon its first use. Our compiler inserts explicit link turn-off instructions (calls) in each

[1] Our approach can also be used with other types of architectures. We will elaborate on this issue later in the paper.

[2] Unless a confusion occurs, we use the terms "node" and "processor" interchangeably in our discussion.

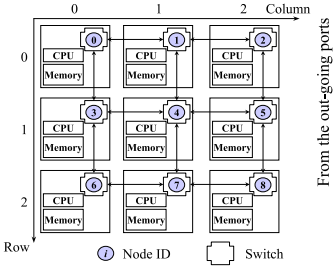

Figure 4: A 3×3 mesh network architecture.

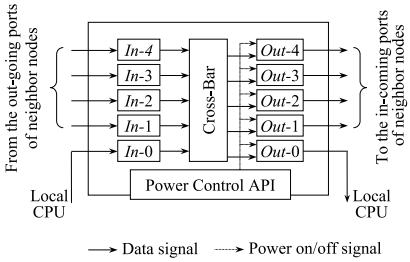

Figure 5: The structure of a switch.

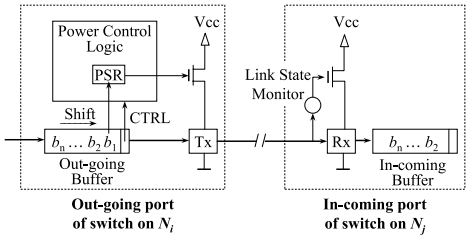

Figure 6: The structure of a link. PSR: power state register. The out-going buffer contains a power control message with power control bit vector $(b_1 b_2 ... b_n)$. The first bit, b_1 is shifted out of the buffer and stored in PSR. The rest of this vector, $(b_2 ... b_n)$, is transferred to the in-coming port of this link, and is subsequently forwarded to the next link.

loop nest to turn-off the communication links whose last usage has taken place.

Before inserting the link turnoff instructions, we first break large loop nests in the given program into a set of smaller loop nests (referred to as "sub-nests") such that the sets of links used by each pair of consecutive sub-nests are different. For a loop nest that uses different communication links during different sets of iterations, splitting it into sub-nests such that each pair of successive sub-nests use different sets of links increases the opportunities for link turnoff. Further, splitting large loop nests into smaller sub-nests allows us to turn off links earlier since we can now turn off links at the end of each sub-nests, instead of waiting for the entire loop nest to terminate.

3.1 Splitting Loop Nests

We now present our compiler algorithm for splitting a loop nest $\mathcal{L}$ into a set of sub-nests $\{\mathcal{L}_1, \mathcal{L}_2, ..., \mathcal{L}_n\}$. These sub-nests are executed in the order of $\mathcal{L}_1, \mathcal{L}_2, ..., \mathcal{L}_n$ so that the order in which the iterations of $\mathcal{L}$ are executed (which are now distributed into sub-nests) is not changed.[3] Assuming that the sets of links used in the loop nests $\mathcal{L}_i$ and $\mathcal{L}_{i+1}$ are $LK(\mathcal{L}_i)$ and $LK(\mathcal{L}_{i+1})$, respectively, we can turn off the links in the set $LK(\mathcal{L}_i) - LK(\mathcal{L}_{i+1})$ to conserve energy without significantly degrading the performance of the application since the links we turn off are not used in the following loop nests. Our goal is to split a given loop nest $\mathcal{L}$ such that we can keep more links in the power-off mode for longer period. In addition, since increasing the number of loop nests increases the size of the application (which may have an adverse impact on code memory management), we split a loop nest only at certain points such that each pair of successive sub-nests use different sets of links. Note that, if two successive sub-nests use the same set of links, we cannot turn off any link at the end of the first sub-nest. Our approach handles each loop nest in the application code one by one, and in the following discussion, we explain our approach for nest $\mathcal{L}$.

We focus on a loop nest $\mathcal{L}$, which can be expressed in an abstract form as follows:

$$\mathcal{L}\text{: for } \vec{I} \in [\vec{L}, \vec{U}] \text{ \{ Body \}}$$

In this loop nest, $\vec{L}$ and $\vec{U}$ are the lower and upper bound vectors, and $\vec{I}$ is the iteration vector.[4] In this paper, we use $[\vec{L}, \vec{U}]$, $(\vec{L}, \vec{U}]$, $[\vec{L}, \vec{U})$, and $(\vec{L}, \vec{U})$ to denote the sets $\{\vec{I} \,|\, \vec{L} \preceq \vec{I} \preceq \vec{U}\}$, $\{\vec{I} \,|\, \vec{L} \prec \vec{I} \preceq \vec{U}\}$, $\{\vec{I} \,|\, \vec{L} \preceq \vec{I} \prec \vec{U}\}$, and $\{\vec{I} \,|\, \vec{L} \prec \vec{I} \prec \vec{U}\}$, respectively, where $\preceq$ and $\prec$ denote lexicographic ordering on vectors.

[3]That is, loop splitting does not affect any data dependence in the loop nest, i.e., it is always legal.

[4]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.

Let us assume that $S(\mathcal{L}) = \{s_1, s_2, ..., s_n\}$ is the set of message-sending instructions in the body of loop nest $\mathcal{L}$. Each message-sending instruction, s_i, has the form "send($d_i(p)$, m)", where m is the message to be sent, p is the id of the node that executes this loop nest, and function $d_i(p)$ gives the id of the destination node for message m. s_i uses connection $C(p, d_i(p))$. Note that, $d_i(p)$ is an invariant since p cannot be changed after we assign the loop nest to a node. It is possible that, in the application code we optimize, there might exist loop nests that the destination node for a send instruction is not an invariant. In this case, we do not optimize the loop nest under consideration.

The set of nodes to which node p sends message during the execution of loop nest $\mathcal{L}$ can be expressed as:

$$D(p, \mathcal{L}) = \{d_i(p) \,|\, s_i \in S(\mathcal{L})\}.$$

We use X_i to denote the set of iterations at which s_i is executed. If the execution of s_i does not depend on any conditional branch instruction, we have X_i equal to the set containing all iterations of loop nest $\mathcal{L}$. In this paper, we only consider X_i sets that can be expressed using Presburger expressions [9].[5] The set of iterations at which connection $C(p, d)$ (where $d \in D(p, \mathcal{L})$) is used can be computed as:

$$U(p, d) = \bigcup_{d_i(p)=d} X_i.$$

Since X_i can be expressed using Presburger expressions, $U(p, d)$ can also be expressed in terms of Presburger expressions. We define $\vec{I}_{min}(p, d)$ and $\vec{I}_{max}(p, d)$ as the first and last iteration vectors for connection $C(p, d)$; i.e., they can be computed as:

$$\vec{I}_{min}(p, d) = \max_{\vec{I} \in U(p,d)} \vec{I},$$
$$\vec{I}_{max}(p, d) = \max_{\vec{I} \in U(p,d)} \vec{I}.$$

Now, we can define the splitting set for loop $\mathcal{L}$ as:

$$SP(p, \mathcal{L}) = \{\vec{I}_{min}(p, d) \,|\, d \in D(p, \mathcal{L})\} \cup \{\vec{I}_{max}(p, d) \,|\, d \in D(p, \mathcal{L})\} \cup \{\vec{L}, \vec{U}\}.$$

We further assume that:

$$SP(p, \mathcal{L}) = \{\vec{I}_1, \vec{I}_2, ..., \vec{I}_n\},$$

where $n = |SP(p, \mathcal{L})|$ and $\vec{I}_1 \prec \vec{I}_2 \prec ... \prec \vec{I}_n$. Since $\vec{L}, \vec{U} \in SP(p, \mathcal{L})$, we have $\vec{I}_1 = \vec{L}$ and $\vec{I}_n = \vec{U}$. Using the vectors in

[5]Presburger formula is a class of logical formulas which can be built from affine constraints over integer variables, the logical connectives ($\vee$, $\wedge$, and $\neg$), and the existential and universal quantifiers ($\exists$ and $\forall$). The Omega Library is an example tool that manipulates integer tuple relations and sets, which are described using Presburger formulas.

splitting set $SP(p, \mathcal{L})$, we can now split loop nest $\mathcal{L}$ into $n-1$ smaller loop nests (sub-nests) as follows:

$$\begin{array}{ll} \mathcal{L}_1: & \text{for } \vec{I} \in [\vec{I}_1, \quad \vec{I}_2] \text{ \{ Body \}} \\ \mathcal{L}_2: & \text{for } \vec{I} \in (\vec{I}_2, \quad \vec{I}_3] \text{ \{ Body \}} \\ & \ldots \ldots \\ \mathcal{L}_n: & \text{for } \vec{I} \in (\vec{I}_{n-1}, \quad \vec{I}_n] \text{ \{ Body \}} \end{array}$$

We handle the conditional statements that may occur within $\mathcal{L}$ conservatively. Specifically, if a message statement can access a communication link depending on the runtime value of some condition, we conservatively assume at compile time that the said statement will access that link. This assumption helps us reduce the potential performance penalty associated with link turn-offs.

3.2 Inserting Link Turnoff Instructions

Our compiler inserts calls (instructions) between each pair of successive loop nests, $\mathcal{L}_i$ and $\mathcal{L}_{i+1}$, to turn off the communication links that have been used in $\mathcal{L}_i$ and will not be used in $\mathcal{L}_{i+1}$. The set of links that are used in a given loop nest $\mathcal{L}_i$ running on node p can be determined as follows:

$$LK(p, \mathcal{L}_i) = \bigcup_{d \in D(p, \mathcal{L}_i)} C(p, d),$$

where $D(p, \mathcal{L}_i)$ is the set of nodes to which node p sends message within loop nest $\mathcal{L}_i$. For a pair of consecutive loop nests, $\mathcal{L}_i$ and $\mathcal{L}_{i+1}$, the set of communication links that can be turned off after $\mathcal{L}_i$ is: $LK(p, \mathcal{L}_i) - LK(p, \mathcal{L}_{i+1})$. If $LK(p, \mathcal{L}_i) - LK(p, \mathcal{L}_{i+1}) = \phi$, no link can be turned off after $\mathcal{L}_i$. In this case, we merge loop nests $\mathcal{L}_i$ and $\mathcal{L}_{i+1}$ if they were extracted from the same loop nest in the previous step.[6]

Figure 7 gives the pseudo code for our algorithm. This algorithm has two phases. In the first phase, we insert code between each pair of successive loop nests if there are links that can be turned off at that point. In the second phase, we merge successive loop nests if no link turnoff instruction is inserted between them.

In our algorithm, we generate link turnoff calls between loop nest $\mathcal{L}_i$ and $\mathcal{L}_{i+1}$ if $LK(p, \mathcal{L}_i) - LK(p, \mathcal{L}_{i+1}) \neq \phi$. The link turnoff instructions turn off the communication links by sending power control messages. Let us assume:

$$D(p, \mathcal{L}_i) = \{d_1, d_2, ..., d_n\}.$$

We send power control messages m_1, m_2, ..., m_n to nodes d_1, d_2, ..., d_n, respectively, to turn off the links in the set $LK(p, \mathcal{L}_i) - LK(p, \mathcal{L}_{i+1})$. Each message m_i carries a link control vector that specifies the power state for each link in the connection from node p to d_i. To each node $d_i \in D(p, \mathcal{L}_i)$, the power control message m_i turns off the links in the following set:

$$F(p, d_i) = C(p, d_i) - (LK(p, \mathcal{L}_{i+1}) \cup \bigcup_{j=i+1}^{n} C(p, d_j)).$$

Since we have:

$$LK(p, \mathcal{L}_i) - LK(p, \mathcal{L}_{i+1}) = \bigcup_{i=1}^{n} F(p, d_i),$$

control messages $m_1, m_2, ..., m_n$ together turn off all the links in $LK(p, \mathcal{L}_i) - LK(p, \mathcal{L}_{i+1})$. Message m_i does not turn off links in $C(p, d_i) \cap LK(p, \mathcal{L}_{i+1})$ because these links will be used in $\mathcal{L}_{i+1}$. Message m_i does not turn off links in $C(p, d_i) \cap C(p, d_j)$ where $i < j \leq n$ either because these links are used by another power control message m_j that will be sent after m_i. Therefore, in the link control vector for message m_i, only those bits corresponding to the links in $F(p, d_i)$ are set to 1.

[6]This is because in this case loop splitting would only increase code size without any energy benefits.

Input: A program consisting of loop nests $\mathcal{L}_1, \mathcal{L}_2, ..., \mathcal{L}_m$;
Output: A program augmented with explicit link turnoff calls;
// Phase-1
for each pair of consecutive loop nests $\mathcal{L}_i$ and $\mathcal{L}_{i+1}$ {
  if $(LK(p, \mathcal{L}_i) - LK(p, \mathcal{L}_{i+1}) \neq \phi)$ {
    assume $D(p, \mathcal{L}_i) - D(p, \mathcal{L}_{i+1}) = \{d_1, d_2, ..., d_n\}$;
    $S_n = LK(p, \mathcal{L}_{i+1})$;
    for $i = n-1$ to 1 step -1 { $S_i = S_{i+1} \cup C(p, d_{i+1})$; }
    for $i = 1$ to n
      if$(C(p, d_i) - S_i \neq \phi)$ {
        $m.type =$ "CTRL";
        $m.body =$ power_control_vector(p, d_i, S_i);
        insert "send(d_i, m)" to the end of $\mathcal{L}_i$
      }
  }
}
// Phase-2
for each pair of consecutive loop nests $\mathcal{L}_i$ and $\mathcal{L}_{i+1}$ {
  if$(LK(p, \mathcal{L}_i) - LK(p, \mathcal{L}_{i+1}) \neq \phi)$ { merge $\mathcal{L}_i$ and $\mathcal{L}_{i+1}$ }
}

power_control_vector(p, d_i, S_i) {
  assume that a message from node p to d_i is transfered along links $l_1, l_2, ..., l_k$
  for $i = 1$ to k { if$(l_i \in S_i)$ $b_i = 0$; else $b_i = 1$; }
  return vector $(b_1, b_2, ..., b_k)$;
}

Figure 7: Algorithm for inserting link turnoff instructions.

4. CONCLUDING REMARKS

Reducing power consumption of networks is an important optimization goal in many application domains, ranging from large-scale simulation codes to embedded multi-media applications. Most of the prior efforts on network power optimization are hardware based schemes. These schemes are predictive by definition as they control communication link status based on observations made in the past. Since prediction may not be very accurate most of the time, these hardware approaches can result in significant overheads in terms of both performance and power. This paper proposes a compiler-driven approach to link voltage management. In this approach, an optimizing compiler analyzes the application code and identifies the last use of a link at each loop nest. It exploits this information by inserting explicit link turn off instructions after the last use of the link.

5. REFERENCES

[1] 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.

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

[3] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface.* MIT Press, 1994.

[4] P. Gupta, L. Zhong, and N. K. Jha. A high-level interconnect power model for design space exploration. In *Proc. the IEEE/ACM International Conference on Computer-Aided Design*, 2003.

[5] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiling Fortran D for MIMD Distributed-Memory Machines. *Communications of the ACM*, 35(8):66–80, August 1992.

[6] J. S. Kim, M. B. Taylor, J. Miller, and D. Wentzlaff. Energy characterization of a tiled architecture processor with on-chip networks. In *Proc. the International Symposium on Low Power Electronics and Design*, Aug 2003.

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

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

[9] W. Pugh. Counting solutions to Presburger formulas: how and why. In *Proc. the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Orlando, Florida, 1994.

[10] 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.

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

[12] 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*, Oct. 2004.