

Topology Selection for Energy Minimization in Embedded Networks

ABSTRACT

The trend towards distributed, networked embedded systems is changing the way power should be managed. Power consumed by bus and network interfaces now matches if not surpasses that of the CPU and is thus becoming a prime candidate for reduction. This paper explores the energy-efficient bus topologies as a new technique for global power optimization of embedded systems that are interconnected by high-speed serial network-like busses such as FireWire and a new generation of SoC busses. Our grammar-based representation for these networks enables the modeling and facilitates selection of energy-efficient bus topology. Experimental results show 15-20% energy saving on the network interfaces without sacrificing system performance.

1. INTRODUCTION

A recent trend in power-aware designs is *communication centric* power management. In both embedded systems and system-on-chip (SoC) architectures, much of the research work in the past decade has gone into making the CPU very power efficient, and the CPU is now consuming a much smaller fraction of the system power. At the same time, bus and network interfaces are consuming the same if not more power. Higher level integration helps alleviate the situation somewhat, but even processors with built-in network interfaces often require two supply voltages: a lower voltage for the core, and a higher voltage for the off-chip I/O. System-on-chip architectures will also face similar issues, as IP components are increasingly being integrated using on-chip networks for power and modularity advantages.

Communication-centric power management schemes can be divided into *custom protocols* vs. *standard protocols*. Custom protocols that utilize application-specific coding schemes [18, 15, 13, 9, 2], custom bus voltages [14], or custom bus segmentation schemes [21, 12, 6] can potentially achieve much better energy efficiency, but they are applicable mainly to closed systems. Most embedded systems and IP components must be interoperable with existing standards, and this limits the types of optimization possible. This paper does not attempt to propose a new standard to compete against the more established ones [3, 7, 8, 10, 11]; instead, it is

intended to demonstrate how an existing standard can incorporate energy efficient optimizations. Some of the most important parameters include communication speed and bus topology. This paper investigates topology selection for FireWire, a hot-pluggable, low-power, high-speed serial bus that can support real-time streaming (isochronous) and asynchronous transfer modes. It is widely available on many embedded systems and computers today.

FireWire bus architecture requires a tree topology. Furthermore, each FireWire component has a limited number of ports and a maximum transfer speed available. Our approach to achieve energy reduction is a grammar-driven, constraint-based searching process for low energy network topology. The advantages are: a) the formal method is a systematic way of modeling and generating topologies; b) our technique is extensible to other buses/networks and is beneficial to system-on-chip design with on-chip networks; c) it is orthogonal to most of the existing CPU-centric power management techniques, thus enabling additive energy savings by combining our techniques with existing ones. Our experimental results show up to 15% to 20% energy savings for network interfaces without sacrificing system performance.

This paper is organized as follows. Section 2 provides FireWire backgrounds and reviews related work. Section 3 presents a formal problem formulation, while Section 4 describes the algorithms we used to search optimal tree topologies. We discuss the experimental results in Section 5. Finally, we conclude our work in Section 6.

2. BACKGROUND AND RELATED WORK

2.1 FireWire Bus

FireWire (IEEE1394) [4] is a high-speed serial bus standard. 1394a currently supports transmission speeds up to 400Mbps, and the new 1394b standard [1] will support transmission speeds of 800Mbps and 1600Mbps. FireWire was designed to connect a computer to peripherals like hard disks, scanners, and consumer electronics like video cameras. It is now widely available on many computers, set-top boxes, and embedded systems in automotive and aerospace domains. FireWire supports two data transfer types: asynchronous and isochronous transfer modes. Asynchronous mode guarantees the data delivery with acknowledgment. Isochronous mode guarantees data bandwidth without acknowledgment, and it is suitable for real-time streams such as like video.

FireWire is hot-pluggable and can connect up to 63 devices. 1394a cables can run as long as 4.5 meters, and packets can take up to 16 hops for a maximum total distance of 72 meters. Future standard extends the single hop distance to up to 100 meters and use fiber optics as physical media. When a new node is attached to the bus,


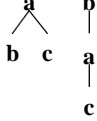
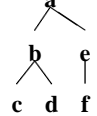
string	tree
$\mathbf{a}(\mathbf{b})(\mathbf{c})$	
$\mathbf{b}(\mathbf{a}(\mathbf{c}))$	
$\mathbf{a}(\mathbf{b}(\mathbf{c}(\mathbf{d}))(\mathbf{e}(\mathbf{f})))$	

Figure 1: Examples of tree strings.

or an existing node is unplugged, the bus will go through bus reset. First, a root will be elected, followed by tree identification and self identification processes, after which the new topology map and speed map is broadcast to every node. Unlike the Universal Serial Bus (USB), which is host-based, FireWire is peer-to-peer.

FireWire imposes a number of restrictions. First, the network must be acyclic. This implies that there is a unique path between any pair of communicating nodes. Second, all intermediate nodes on the path must be powered on (at least the physical layer controller) to act as repeaters. Third, all the intermediate nodes must support the transfer speed of the communicating nodes, otherwise the transaction cannot be started. Fourth, the fan-out of each node is constrained by the number of ports available on the physical interface.

2.2 Power Management with FireWire

Power management opportunities with a standard protocol like FireWire are at higher level than most previous works. Circuit-level bus voltage scaling techniques, including [14, 20], which make the bus voltage and frequency track the bus traffic, or voltage swing reduction [17], would not be applicable due to interoperability reasons. Bus coding that minimizing the transition activities on buses [18, 15, 13, 9, 2] would not be applicable, either. Buses segmentation to reduce bus load and improve latency [21, 12, 6] may be applicable in principle, but they must be adapted to the specific capabilities of the bus standard. Our technique is similar to bus segmentation in the sense that both try to localize the bus traffic so that the high-cost global bus activities are minimized. While traditional bus segmentation techniques mainly partition and cluster the bus nodes into segments, our approach works with the constraints imposed by the bus standard on the topology, port count, and transfer speed. To accomplish this, we model the legal topologies using a tree grammar, and we use the constraints to prune the search space. We present an algorithm that finds a topology that minimizes total energy consumption for the same communication traffic. The experimental results are validated using a FireWire snooter.

3. PROBLEM FORMULATION

We generate tree topologies for FireWire by incrementally attaching new nodes to existing trees. We have developed a formal representation for modeling trees and generating tree topologies. In this section we give several definitions, followed by the cost function and our problem statement.

3.1 Definitions

Definition 1 (Node $\mathbf{u} \in U$) A node \mathbf{u} is a component in the system that has bus interfaces ready to connect to other components. $p_{\mathbf{u}}$ is the number of ports available for \mathbf{u} . $S_{\mathbf{u}}$ is a finite set of speeds that node \mathbf{u} can work at.

Definition 2 (Tree) A tree is a connected component $C \subseteq U$ with exactly $|C| - 1$ undirected edges.

Definition 3 (Transaction $\tau \in \Gamma$) A transaction $\tau = (\mathbf{u}_1, \mathbf{u}_2, s, w)$ is a data transfer behavior between two nodes \mathbf{u}_1 and \mathbf{u}_2 at the transfer speed s with non-zero workload w , where $s \in S_{\mathbf{u}_1} \cap S_{\mathbf{u}_2}$ and w is the amount of data (in byte) transferred.

We require the transactions in the system are peer-to-peer. Multicast or broadcast transactions are not allowed.

Definition 4 (Tree string t) A tree string is a string representation of a tree. It is obtained by in-order traversal of the tree. The root of the tree is traversed first, then recursively each child is traversed. For example, in Figure 1, the string $\mathbf{a}(\mathbf{b})(\mathbf{c})$ represents a tree of three nodes, with \mathbf{a} the root node and \mathbf{b} and \mathbf{c} leaf nodes. A matched pair of parentheses with the substring inside represents a subtree. The string $\mathbf{a}(\mathbf{b}(\mathbf{c}(\mathbf{d}))(\mathbf{e}(\mathbf{f})))$ represents a tree of six nodes. \mathbf{c} and \mathbf{d} are two subtrees (also leaf nodes) of \mathbf{b} , and $\mathbf{b}(\mathbf{c}(\mathbf{d}))$ and $\mathbf{e}(\mathbf{f})$ are two subtrees of \mathbf{a} .

Definition 5 (Tree grammar G) Let Σ be an alphabet $\Sigma = \{\mathbf{u} | \mathbf{u} \in U\} \cup \{(\ ,)\}$, and a node \mathbf{u} is denoted by a lower-case Roman letter. A tree is represented by a tree string t that can be generated from grammar $G = (V, \Sigma, P, S)$, where $V = \{B, E\}$ is a set of variables, S is a start symbol, P is a set of productions $V \rightarrow V \cup \Sigma$:

$$\begin{aligned} E &\rightarrow \mathbf{u} \\ B &\rightarrow (E) \\ E &\rightarrow EB \\ S &\rightarrow E \end{aligned}$$

and if a node \mathbf{u} appears in t , it appears exactly once.

Definition 6 (Tree language L) A language $L(\Sigma) = \{t | t \text{ is in } \Sigma^* \text{ and } S \Rightarrow t\}$ is a set of tree strings generated by grammar G . We also use $L(v) = \{t | t \text{ is in } \Sigma^* \text{ and } v \Rightarrow t\}$ to denote the set of strings generated with the start symbol $v \in V$, and $L(v^*) = \{t^* | t \text{ is in } \Sigma^* \text{ and } v \Rightarrow t\}$ to denote the set of strings that has zero or one or more concatenated substrings each of which is generated with a start symbol $v \in V$.

Let $|t|$ represent the length of the tree string t . It is easy to see that for a tree containing n nodes, $|t| = 3n - 2$.

A tree topology can be represented by multiple tree strings. For example, $\mathbf{a}(\mathbf{b})(\mathbf{c})$ and $\mathbf{b}(\mathbf{a}(\mathbf{c}))$ in Figure 1 represent the identical topology with different roots. Even with the same root, tree string $\mathbf{a}(\mathbf{b})(\mathbf{c})$ and $\mathbf{a}(\mathbf{c})(\mathbf{b})$ represent the same tree. Since any node (capable of bus management) on a FireWire bus can be the root, we can pick one node as the root and order the rest so that we are able to obtain a canonical form of a tree string.

Definition 7 (Transforming function H) A transforming function H converts a tree string to its canonical form by the means of in-order traversal with sorting of labels. The canonical form of a tree

string t is: $\forall \mathbf{u}$ in t , \mathbf{u} and its all children are sorted in a lexicographical order. Tree string $t_1 = \mathbf{a}(\mathbf{b}(\mathbf{c})(\mathbf{d}))(\mathbf{e}(\mathbf{f}))$ in Figure 1 is in its canonical form. Tree string $t_2 = \mathbf{a}(\mathbf{e}(\mathbf{f}))(\mathbf{b}(\mathbf{c})(\mathbf{d}))$ is not in its canonical form since \mathbf{a} and its children \mathbf{e} and \mathbf{b} are not sorted. Thus we have $t_1 = H(t_2)$.

New trees can be formed by adding a node x to an existing tree. The node can be either attached as a leaf node or inserted as a non-leaf node. We define a growing pattern $F(t, x)$ to help incrementally generate larger trees from smaller ones.

Definition 8 (Growing function F) $L(\Sigma \cup \{x\}) = L(\Sigma) \cdot F(t, x)$, for all $t \in L(\Sigma)$. Tree strings in $L(\Sigma \cup \{x\})$ can be derived from trees in $L(\Sigma)$ according to the following rules:

$$F(t, x) = \begin{cases} \mathbf{d}(x) & \text{if } t = \mathbf{d}, \\ \mathbf{d}(x)(\beta)\gamma \cup \mathbf{d}(x(\beta))\gamma \cup \\ \mathbf{d}(F(\beta, x))\gamma \cup \mathbf{d}(\beta)F'(\gamma, x) & \text{if } t = \mathbf{d}(\beta)\gamma. \end{cases} \quad (1)$$

$$F'(\alpha, x) = \begin{cases} \emptyset & \text{if } \alpha = \epsilon, \\ (F(\beta, x))\gamma \cup (\beta)F'(\gamma, x) & \text{if } \alpha = (\beta)\gamma. \end{cases} \quad (2)$$

where $\mathbf{d} \in U$ represents the root of tree t , $\beta \in L(E)$ and $\gamma \in L(B^*)$.

Definition 9 (Tree string set T) A tree string set T for a node set U is a set of tree strings generated by grammar G and:

- 1) $\forall t \in T, \forall u \in U, u$ is in t ,
- 2) $\forall t \in T, t = d_0 D, d_0 \in U, D \in L(B^*)$,
- 3) $\forall t_1, t_2 \in T$, if $t_1 \neq t_2$, then $H(t_1) \neq H(t_2)$, and
- 4) for any tree string $t = d_0 D, d_0 \in U, D \in L(B^*)$, $|t| = 3|U| - 2$, $\exists t' \in T, H(t) = H(t')$.

In other words, each tree string in T contains all nodes in U . All the tree strings have the same root node d_0 . No two tree strings in T have the same canonical representation. Tree set T represents a complete set for all the tree topologies for the node set U .

We assume that all the nodes in U are connected to form a single tree topology. A forest consisting of multiple trees is not allowed.

Lemma 1 (Tree generation) Given a tree string set T for a node set U , a new tree string set T' for the node set of $U \cup \{x\}$ is derived from T without producing identical topologies by applying growing function F to each tree in T : $T' = T \cdot F(t, x)$, for all $t \in T$.

Due to paper length limitation, the proof of Lemma 1 is omitted. Please refer to [16] for the details.

Definition 10 (Port count constraint) A tree can be represented in the form of \mathbf{dB}^k , where \mathbf{d} represents the root of the tree, and $B^k (B \in L(B))$ represents all of its k subtrees. The port count constraint is: $\forall u \in U$,

$$\begin{cases} p_u \geq k + 1 & \text{if } p \text{ is a non-root node,} \\ p_u \geq k & \text{if } p \text{ is the root node.} \end{cases} \quad (3)$$

where p_u is the port count for node u .

For example, in Figure 1, the tree string $\mathbf{a}(\mathbf{b}(\mathbf{c})(\mathbf{d}))(\mathbf{e}(\mathbf{f}))$ satisfies (3) if $f_a, f_e \geq 2$, $f_b \geq 3$ and $f_c, f_d, f_f \geq 1$.

Corollary 1 (Connectivity condition) Given a node set U of n nodes, a tree topology that connects all the nodes exists, iff

$$\sum_{u \in U} p_u \geq 2n - 2 \quad (4)$$

where p_u is the port count of node u .

A tree is *legal* if every node in U satisfies the port count constraint (3) and connectivity condition (4).

3.2 Cost Function

Given a transaction $\tau = (u_\tau, v_\tau, s_\tau, w_\tau)$, all the nodes $u \in U$ can be categorized into three sets: M_t, M_r , and M_i . $M_t = \{u_\tau, v_\tau\}$ consists of communicating nodes. M_r consists of all the nodes that repeat the transaction τ on the routing path. M_i consists of the nodes not involved in the transaction τ . We say the working modes m_u for the node u in the above sets each are transferring, repeating, and idle, respectively.

For a given node u , the power function P is a function of the port number p_u and working mode m_u , denoted as $P(p_u, m_u)$. Power function can be a lookup table whose data entries come from manufacture's data sheets [19].

We define the power function of a transaction τ and a tree t as:

$$P(\tau, t) = \sum_{u \in M_t} P(p_u, m_u) + \sum_{u \in M_r} P(p_u, m_u) \sum_{u \in M_i} P(p_u, m_u) \quad (5)$$

Power function $P(\tau, t)$ represents the total bus power of the whole systems during the transaction τ . It consists of power of nodes involved the transaction (both transferring and repeating node) and power of idle nodes.

For a transaction τ , *Effective transaction time* is defined as:

$$D_\tau = \frac{w}{s}. \quad (6)$$

where w is the workload and s is the transmission speed.

Note that effective transaction time may not be equal to the actual time to complete a transaction. Consider two transactions start at the same time, both transferring data at the same speed and both taking one minute to complete. Assume they equally share the total bandwidth, the effective transaction time for each transaction is only half a minute.

During a given time period D , we suppose there are k transaction instances $\{\tau_i\} (i = 1, \dots, k)$. The total effective transaction time is:

$$D_\tau = \sum_i D_{\tau_i}. \quad (7)$$

We define *utilization* of the transaction τ is:

$$\lambda_\tau = \frac{D_\tau}{D}. \quad (8)$$

Finally for a given tree string t , we define our cost function as:

$$C = \sum_{\tau \in \Gamma} P(\tau, t) \lambda_\tau \quad (9)$$

Cost C represents the average energy consumption on the bus in unit time. However it does not include the energy consumption when the bus is completely idle (no transaction occurs).

```

TreeGen( $V, \Gamma, h$ )
0 # input : node set  $U$ , transaction set  $\Gamma$ , hub type  $h$ 
1 # output: tree set  $T$ 
2 # Preprocess: add hub nodes if necessary
3  $V' \leftarrow$  preprocess( $V, h$ ), #sort nodes in decreasing order by their  $p_u$ .
4 for each  $v$  in  $U' \setminus \{p[v] \leftarrow p_u\}$  #  $p[v]$ : port count of  $V$ .
5  $v \leftarrow$  pop up the first node in  $U'$ 
6  $T \leftarrow \{u\}$ 
7 while  $U'$  not empty {
8    $u \leftarrow$  pop up the first node in  $U'$ 
9    $T' \leftarrow T$ 
10  for each tree  $t$  in  $T$  {
11    for each node  $v$  in  $t$  {
12       $T_l \leftarrow$  AddAsLeaf( $t, u, \Gamma$ )
13       $T_b \leftarrow$  AddAsBranch( $t, u, \Gamma$ )
14       $T' \leftarrow T_l \cup T_b$ 
15    }
16  }
17   $T \leftarrow T'$ 
18 }
19 return  $T$ 

```

Figure 2: The tree enumeration algorithm.

3.3 Problem Statement

Given a tree t and a set of transactions Γ , the tree is a *feasible* one if it satisfies the speed constraint:

$$\forall \tau(u_\tau, v_\tau, s_\tau, w_\tau) \in \Gamma \text{ and } \forall x \in M_r, s_\tau \in S_x. \quad (10)$$

That is, for a transaction, all the intermediate nodes on a routing path should support the transfer speed. We aim to find trees that has the minimum cost defined by (9). The input to the problem is a set of node V and a set of transaction Γ . The output of the problem is a tree (or a set of trees) with minimum cost.

In case the input node set U does not satisfies the connectivity condition (4), we add *hubs* to interconnect the nodes so that the connectivity condition is satisfied. A hub is a special node that can repeat transactions but cannot be a peer node in a transaction. Several types of hubs are available, which differentiate by their port counts. The more ports a hub has, the more power it consumes when repeating packets. We are also interested in finding out which hub type is energy-optimal in connecting the node devices.

4. ALGORITHM

Tree topologies are incrementally generated using our grammar-based growing function. In this section we present the tree generation algorithm and the top level search algorithm. A brief discussion on complexity shows that asymptotically our algorithm generate much fewer trees than exhaustive approach and in practice, our technique produce even much fewer trees by applying system-level constraints.

4.1 Approach

We take an incremental approach to obtain the tree set of $k+1$ nodes from a tree set of k nodes. We use our growing function F to add a node to an existing tree either as a leaf node or as a non-leaf node. At each incremental step, if a tree topology fails to satisfy the port count constraint or the transfer speed constraint, it will not be included into the tree set. After obtaining a tree set for all the nodes, we calculate cost for each tree to search for optimal topologies.

4.2 Algorithms

```

AddAsLeaf( $t, x, \Theta$ )
1 # input : tree  $t$ , node  $x$ , transaction set  $\Gamma$ 
2 # output: tree set  $T_l$ 
3  $T_l \leftarrow \emptyset$ 
4  $ptr \leftarrow 0$ 
5 while  $ptr < len(t)$  {
6   while  $t[ptr] \notin D$  {  $ptr \leftarrow ptr + 1$  } # find next node id
7   if  $p[t[ptr]] > 0$  { # if port available
8      $T_{sub} \leftarrow$  Subtree( $t[ptr]$ ) #  $T_{sub}$ : a set of subtrees of  $t[ptr]$ 
9     insertx( $T_{sub}, x$ ) # so that elements in  $T_{sub}$  are sorted
10     $t' \leftarrow$  join( $T_{sub}$ ) # concatenate elements in  $T_{sub}$  into a string
11     $t'' \leftarrow$  insertSub( $t, t'$ ) # substitute  $t[ptr]$ 's subtrees for  $t'$ 
12    updatePort( $p$ ) # update port count information
13     $tag \leftarrow 1$ 
14    for each  $\tau$  in  $\Theta$  {
15      if checkSpeed( $t'', \tau$ ) == FALSE {
16         $tag \leftarrow 0$ ; break }
17    }
18    if  $tag == 1$  {  $T_l \leftarrow T_l \cup \{t''\}$  }
19  }
20   $ptr \leftarrow ptr + 1$ 
21 }
22 return  $T_l$ 

```

Figure 3: The AddAsLeaf routine.

```

MinTree( $V, \Theta, H$ )
1 # input : node set  $V$ , transaction set  $\Theta$ , hub type set  $H$ 
2 # output: optimal tree set  $minTreeSet$ , minimum cost  $minCost$ 
3  $minTreeSet \leftarrow \emptyset$ 
4  $minCost \leftarrow \infty$ 
5 for each  $h$  in  $H$  {
6    $T \leftarrow$  TreeGen( $V, h$ )
7   for each  $t$  in  $T$  {
8      $cost \leftarrow$  getCost( $t, \Theta$ )
9     if  $cost < minCost$  {
10       $minCost \leftarrow cost$ ;  $minTreeSet \leftarrow \{(t, h)\}$ 
11     }
12     else if  $cost == minCost$  {
13        $minTreeSet \leftarrow minTreeSet \cup \{(t, h)\}$ 
14     }
15   }
16 }
17 if  $minCost < \infty$  {print  $minCost, minTreeSet$  }
18 else { print 'no solution found' }

```

Figure 4: The top level algorithm.

The tree generation algorithm is shown in Figure 2. The inputs to the algorithm are a node set U and hub type h . The output of the algorithm is a tree set containing all the feasible trees. In the pre-process procedure in Line 3, we check whether the node set U satisfies the connectivity condition (4). If port count is not enough, we add adequate number of hubs of type h into the node set, thus forming a new node set U' . We also sort the nodes in U' by their port counts to facilitate the tree generation described below. Array $p[n]$ (Line 4) keeps the port count information of all nodes in U' during the process of tree generation. Line 5-6 gets the first node in U' and initialize the tree set T . The while loop (line 7-18) incrementally generates new trees and expands tree set. Two main steps are *AddAsLeaf()* and *AddAsBranch()* which add a new node to the existing tree set as a leaf node and as a non-leaf node, respectively.

Figure 3 shows the procedure *AddAsLeaf()*. When adding a new node x to an existing tree t as a leaf node, we try attaching x to each node if it has a port available. In the string of tree t , we insert (x) after a node u to the right position so that the new tree remains in its canonical form. We identify the routing path between two communicating nodes u and v , check speed constraints for every intermediate nodes (if any), and return whether the tree satisfies the speed constraint. If for all transactions, the tree satisfies speed constraints, we append it to the tree set.

The other step, similar to *AddAsLeaf()*, is to add x as a branch node. A connection between a node u and one of its subtrees is identified. x is inserted as the child of u while the subtree as the child of x . Thus x becomes a non-leaf node. We repeat this for all the subtrees of u . The procedure *AddAsBranch()* is implemented similarly to *AddAsLeaf()* as string manipulation and it not shown.

A top level algorithm is shown in Figure 4. We assume the connectivity condition (4) is not satisfied thus we try different types of hubs in the outmost loop (line 5 and 15). Otherwise the loop can be safely removed. Line 6 generates all feasible trees and stores them in set T . In the loop of line 11-19, we check to see whether the speed constraint is satisfied. If yes, we then calculate its cost and save it if it is minimum cost. Finally we output the optimal tree set with hub type or no solution message if all topologies fails to satisfy the constraints.

4.3 Complexity

Let us first see the complexity of an exhaustive approach. Given a node set U of n nodes, we can permute the nodes and obtain $n!$ strings, each consisting of n nodes. For each string, we need to add $n - 1$ pairs of parenthesis to form a tree string. For each parentheses pair, we have $n - 1$ locations to add, and adding parentheses pairs is independent with each other. Thus we have 2^{n-1} of ways to add $n - 1$ pairs of parentheses. Altogether, we can obtain $n!2^{n-1}$ trees from a node set of n nodes. Note that among those trees, there are trees that topologically identical but differ in root nodes, trees that are not in their canonical forms, and trees that do not satisfy constraints.

Our algorithm assumes a node to be the root node and trees differ only in the root will not be repetitively generated. Our algorithm generates tree strings in their canonical forms and does not generated topologically identical tree strings.

In the *AddAsLeaf()* routine, the if-branch (line 8 – 18) produces at most k new strings (k is the number of nodes in current tree t). *AddAsBranch()* routine produces at most $(k - 1)$ new strings. Thus

Port/mode	Transfer	Repeat	Idle
1	158.4	138.6	125.3
2	234.3	217.7	174.7
3	379.5	320.1	247.5
4	676.5	498.3	412.5
6	924.0	673.2	541.2

Table 1: Power data of FireWire interface (in mW).

Device	Max speed(Mbps)	port #
Mac1	400	2
Mac2	400	2
PC1	400	2
HD1	200	2
Cam	100	1
iBot1	200	1
ibot2	200	1
Hub	400	3/4/6

Table 2: A list of FireWire devices

we obtains at most $(2k - 1)$ tree strings for a tree of $(k + 1)$ nodes. Theoretically, our algorithm may produce at most $(2n - 3)!!$ patterns for a node set of size n , which sets a very loose upper bound of generated tree strings. This is already asymptotically smaller than the exhaustive approach. In reality, our algorithm generates much fewer trees since we apply constraints at each incremental step. This greatly reduces the generated trees in that step and avoids fast growing of trees in the succeeding steps. For example, when $n = 8$, theoretically the exhaustive approach produces 5160960 trees and our algorithm may produce at most 135135 strings, only 2.6% of the former approach. But in reality we only generated as few as 90 trees (see Section 5) thanks to the constraints we applied.

5. EXPERIMENTAL RESULTS

We apply our algorithm to two FireWire bus examples. We use Firebug [5], a software bus snooping tools, to monitor the bus traffic and obtain the workload information. In the first example we have eight nodes to be connected. The second example has the similar setup but in a larger scale, which makes it almost impossible for the exhaustive approach to find out a solution in a practical time period. Our algorithm generates optimal tree sets efficiently. Our experimental results show that the optimal solutions we found save up to 15%-20% energy compared to an arbitrarily generated topology. Furthermore, workload balanceness and hub types have perceivable influences on energy cost. We have built a web-based tool to facilitate the user to interact with the core algorithm on the server side.

Trans.	u1	u2	Speed(Mb/s)	Workload (x1000Mb)
1	Mac1	HD1	200	13
2	Mac1	PC1	400	25
3	Mac1	Cam	100	80
4	Mac1	iBot1	200	46
5	Mac2	HD1	200	5
6	PC1	iBot2	200	46

Table 3: A list of transactions

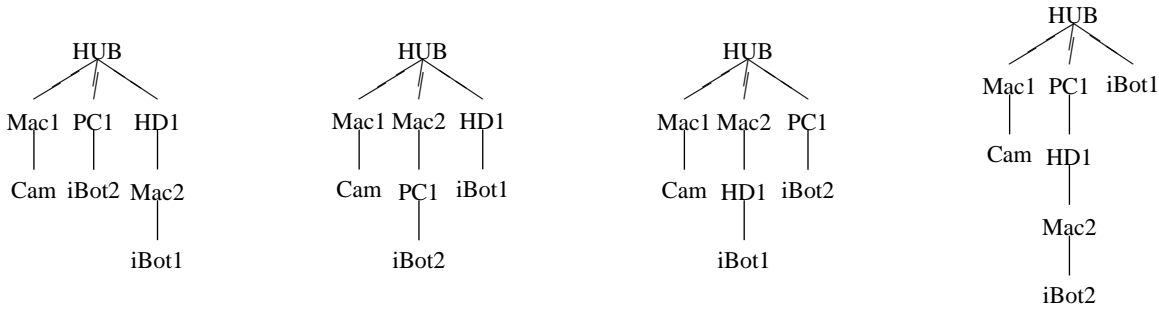


Figure 7: Example I: $p = 3$, four trees found.

Hub type	$p = 3$	$p = 4$	$p = 6$
# of trees	90	269	376
MaxCost	270.6	306.2	338.9
MinCost	213.2	267.5	290.8
diff(%)	12.2	14.5	16.6
# of optimal trees	4	1	1

Table 4: Experiment results for Example I (eight nodes).

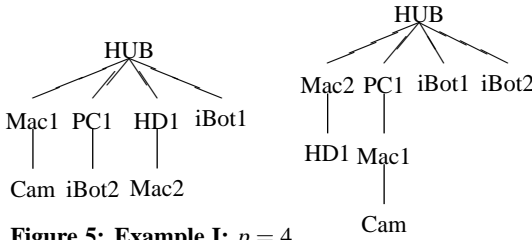


Figure 5: Example I: $p = 4$, one tree found.

Figure 6: Example I: $p = 6$, one tree found.

Example I

We have seven devices to be connected with FireWire bus interfaces, as listed in table 2: Mac1 and Mac2 are two desktop Mac computers, PC1 is a notebook computer, HD1 is a FireWire hard drives, Cam is a Camcorder, iBot1 and iBot2 are two web cameras. A hub is added to the device list in order to satisfy the connectivity condition.

We use FireBug to capture the workload information. FireBug can keep track of all the activity on the FireWire bus and report to the user the desired events by filtering out the irrelevant ones. We first arbitrarily interconnect all the devices and turn on FireBug to monitor and record the traffic on the bus, and extract transaction-related information from FireBug log file. For example, we obtain the nodes involved in a transaction, the data transfer speed and the number of packets transferred. Then we obtain the transaction table shown in Table 3.

Table 4 shows the experiment results. Although this experiment looks simple, to find the optimal solution is not trivial. Exhaustive enumeration will produce 5160960 trees (see the last section). Our algorithm shrinks the tree set sizes down to less than 400 (first line of Table 4) using our grammar-driven tree generation.

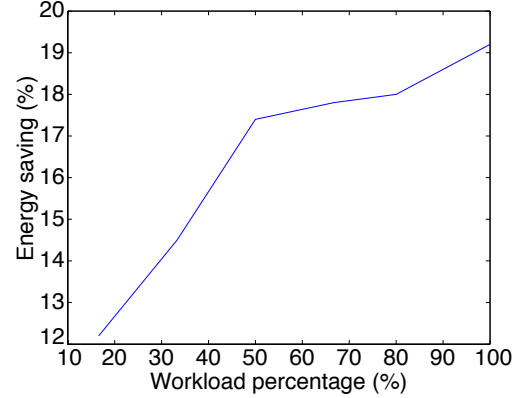


Figure 8: Workload balanceness vs. potential energy saving.

MaxCost and *MinCost* are the maximum and minimum cost value for all generated feasible trees. In three cases ($f_n = 3, 4, 6$), the differences between *MinCost* and *MaxCost* are ranging from 12.2% to 16.6%, representing the potential energy savings by selecting the trees with *MinCost*. It is interesting to see that the more ports the hub has, the more energy the tree consumes. The reason is that the hub with more ports consumes more energy to repeat packets. Therefore for this example, a three-port hub is the optimal solution.

Figure 7, 5 and 6 show the optimal tree sets when using hubs of three, four and six ports, respectively. When using a three-port hub, four trees are found (see Figure 7). When using a six-port hub, only four ports of the hub are used. This is because for some transactions, it costs less when the two peer nodes are directly connected (if possible) instead of going through a hub. Trees in Figure 5 and Figure 6 are different even in both cases four ports are used. The reason is that different hub types in the two cases causes different energy consumption.

In order to see whether the potential energy savings are sensitive to the workload balanceness, we change the workload on transaction 3 (between *Mac1* and *Cam*) and generate optimal topology for each workload value. Transaction 3 originally has the largest workload among all transactions. We change its workload value from the average of all transactions to positive infinity (disabling all other transaction). Figure 8 shows the curve of the workload percentage vs. the potential energy savings. The workload percentage is the ratio between the workload of the selected transaction to the total workload of all transactions. The curve shows that the higher the

hub type	$p = 3$	$p = 4$	$p = 6$
# of hub	3	2	1
# of total devices	13	12	11
# of trees	45761	17001	2013
MaxCost	332.8	304.4	270.4
MinCost	300.9	268.8	236.7
diff(%)	10.1	13.3	14.2
# of optimal trees	3	2	1

Table 5: The number of devices with different hub types.

workload percentage is, the higher the potential energy saving becomes. This implies that in the scenario where more unbalanced workload exists, the potential energy saving becomes more significant, up to nearly 20% in this example.

Example II

We use three Mac computers, four FireWire hard drives, one printer, one scanner and one camcorder, totally ten devices. To satisfy the connectivity condition, we add three, two, and one hub when using three-port, four-port, six-port hubs, respectively (first two rows in Table 5). For the exhaustive approach, the problem of up to thirteen nodes becomes intractable in practice. Our algorithm generated highly compact tree sets (Row 3 of Table 5). Potential energy savings ranges 10.1%-14.2%.

Note that in our cost function, we only consider the time periods when there is traffic on bus. When the bus is complete idle, part or all the bus nodes can be potentially disabled, resulting more energy savings. In the implementation of FireWire bus drivers, the link layer and above layers can be disabled for low power if no transaction is on the node. While in our examples, we assume all the layers are on all the time. Even for the physical layer controller, dynamic power management techniques can be applied to disable it when there is no traffic passing through it. All the above conditions are orthogonal to our techniques. It is foreseeable that additive energy saving could be achieved by combining our technique together with other power management techniques.

6. CONCLUSION

This paper presents a method for optimizing peer-to-peer serial bus topology for energy reduction. We use a canonical string form to represent trees that is both concise and easy to manipulate. We propose an incremental approach to enumerate tree topologies. By applying a number of constraints in each enumeration step, we are able to obtain both complete and compact tree sets without producing redundant trees. We capture the bus workload information by monitoring the bus traffic and factor it into the cost function for energy optimization.

Workload balanceness has an impact on the potential energy savings. The more imbalanced the workload is, the more potential energy savings can possibly be achieved. Hub type selection influences the optimal solution points due to variations in their individual power behavior.

Although we use FireWire bus to demonstrate the effectiveness of our technique, our approach is general enough to apply to many other tree-like architectures. As low power serial buses becomes more popular in computer systems and system-on-chip, we believe our technique can be applied to more applications and will show

significant energy savings.

Current topology optimization is static, requiring the bus to reconfigure at least once to form an optimal topology. It is possible to construct a bus topology with redundant physical links while dynamically configure it to form logic tree topologies for performance, energy-saving, and fault-tolerance reasons.

7. REFERENCES

- [1] 1394 Trade Association. P1394b draft standard for a high performance serial bus (high speed supplement). In <http://www.zayante.com/p1394b/drafts/p1394b1-33.pdf>, 2001.
- [2] Y. Aghaghi, F. Fallah, and M. Pedram. Irredundant address bus encoding for low power. In *Proc. of Int. Symposium on Low Power Electronics and Design*, pages 182–187, August 2001.
- [3] V. Alliance. On chip bus attributes version 1. In <http://www.vsi.org/library/specs/summary.htm>, 1998.
- [4] D. Anderson. *FireWire System Architecture*. MindShare Inc., Reading, Massachusetts, second edition, 1999.
- [5] Apple Inc. Apple's firewire sdk 2.8.1. In ftp://ftp.apple.com/developer/Development_Kits/FireWire_2.8.1_SDK.sit.bin, 2000.
- [6] J. Chen, W. Jone, J. Wang, H.-I. Lu, and T. Chen. Segmented bus design for low-power systems. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, 7(1):25–29, March 1999.
- [7] H. Consortium. Hypertransport I/O link specification 1.03. In http://www.hypertransport.org/downloads/HT_IOLink_Spec.pdf, 2001.
- [8] W. J. Dally and B. Towles. Route packets, not wires: On-chip interconnection networks. In *Proc. of DAC*, pages 684–689, June 2001.
- [9] J. Henkel and H. Lekatsas. A^2BC : adaptive address bus coding for low power deep sub-micron designs. In *Proc. of the 38th Design Automation Conference*, pages 744–749, June 2001.
- [10] IBM. Coreconnect bus architecture. In <http://www.chips.ibm.com/products/coreconnect/index.html>, 1999.
- [11] Intel. Third generation I/O architecture. In <http://developer.intel.com/technology/3GIO>, 2001.
- [12] J. Kim and A. El-Amawy. Performance and architectural features of segmented multiple bus system. In *Proc. of International Conference on Parallel Processing*, pages 154–161, 1999.
- [13] S. Komatsu, M. Ikeda, and K. Asada. Low power chip interface based on bus data encoding with adaptive code-book method. In *Proc. Ninth Great Lakes Symposium on VLSI*, pages 368–371, March 1999.
- [14] L.-S. P. L. Shang and N. Jha. Power-efficient interconnection networks: Dynamic voltage scaling with links. *Computer Architecture Letters*, 1(2), May 2002.
- [15] E. Musoll, T. Lang, and J. Cortadella. Working-zone encoding for reducing the energy in microprocessor address buses. *IEEE Trans. on VLSI Systems*, 6(4):568–572, December 1998.
- [16] Omitted for blind review.
- [17] A. Rjoub, S. Nikolaidis, O. Koufopavlou, and T. Stouraitis. An efficient low-power bus architecture. In *Proc. of IEEE Int. Symposium on Circuits and Systems*, pages 1864–1867, June 1997.
- [18] M. Stan and W. Burleson. Bus-invert coding for low-power I/O. *IEEE Trans. on VLSI Systems*, 3(1):49–58, March 1995.
- [19] Texas Instruments. IEEE 1394 products: Integrated devices, link layer controllers and physical layer controllers. In <http://www.ti.com/sc/1394>, 2002.
- [20] G.-Y. Wei, J. Kim, D. Liu, S. Sidiropoulos, and M. Horowitz. A variable-frequency parallel I/O interface with adaptive power-supply regulation. *IEEE Journal of Solid-State Circuits*, 35(11):1600–1610, November 2000.
- [21] Y. Zhang, W. Ye, and M. Irwin. An alternative architecture for on-chip global interconnect: segmented bus power modeling. In *Conf. Record (Signals, Systems & Computers) of 32nd. Asilomar Conf.*, pages 1062–1065, 1998.