Topology Selection for Energy Minimization in Embedded Networks [*]

Dexin Li, Pai H. Chou, and Nader Bagherzadeh
Dept. of ECE, University of California, Irvine, CA 92697-2625 USA
{dli,chou,nader}@ece.uci.edu

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 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 buses. Our grammar-based representation for these networks enables selection of energy-efficient bus topologies. Experimental results show 15–20% energy saving on the network interfaces without sacrificing system performance.

I. Introduction

A recent trend in power-aware designs is *communication centric* power management. Bus and network interfaces in embedded systems are consuming a significant amount of power. 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*. This paper does not attempt to propose a new standard but is intended to demonstrate how an existing standard can incorporate energy efficient optimizations. More specifically, we investigate topology selection for FireWire busses.

FireWire (IEEE1394) [1] is a high-speed serial bus architecture, supporting two data transfer types: asynchronous and isochronous. It is hot-pluggable and a single bus can connect up to 63 devices. The packets transferred can take up to 16 hops for a maximum total distance of 72 meters. When a new node is attached to the bus, or an existing node is unplugged, the bus will go through a bus reset and automatically reconfigure itself.

In spite of the abundant bus management features, current FireWire busses have implemented very limited power management schemes. We believe that the rich bus management features open up new opportunities in high-level power management. We envision that a centralized bus manager that is aware of the bus topology can optimize for energy reduction based on workload and the speeds of transactions on the bus.

FireWire imposes a number of restrictions. First, the network must be acyclic. Second, all intermediate nodes on the routing 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, Fourth, the fan-out of each node is constrained by the number of ports available on the physical interface. Our approach works with the constraints imposed by the bus standard on the topology, the port count, and the transfer speed. To accomplish this, we model the legal topologies using a tree grammar, and use the constraints to prune the search space. Our experimental results show up to 15% to 20% energy savings for network interfaces without sacrificing system performance.

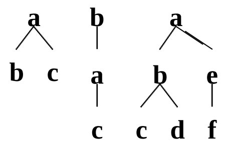

Fig. 1. Examples of tree strings: **a**(**b**)(**c**), **b**(**a**(**c**)), and **a**(**b**(**c**)(**d**))(**e**(**f**)), respectively.

$$S \rightarrow E$$
$$E \rightarrow \mathbf{u}|EB$$
$$B \rightarrow (E)B|\varepsilon$$

Fig. 2. The production set for the tree grammar G.

II. Problem Formulation

We generate tree topologies 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.

A. Definitions

Definition 1 (Node $\mathbf{u} \in U$) A node $\mathbf{u}$ is a component in the system that has an interface consisting of one or more ports 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 at which node $\mathbf{u}$ can operate.

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 process 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 bytes) transfered. [1]

[*]This research was sponsored by DARPA under contract F33615-00-1-1719

[1]We assume all the transactions 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. For example, the string $\mathtt{a(b)(c)}$ in Fig. 1 represents a tree of three nodes, with **a** being the root node and **b** and **c** being leaf nodes. A matching pair of parentheses with the substring inside represents a subtree.

Definition 5 (Tree grammar G) Let Σ be an alphabet $\Sigma = \{\mathbf{u}|\mathbf{u}\in U\} \cup \{\mathtt{(},\mathtt{)}\}$, and a node **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 $V \to V \cup \Sigma$ shown in Fig. 2, where ε is an empty string, and if a node **u** appears in t, it appears exactly once.

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

A tree topology can be represented by multiple tree strings. For example, $\mathtt{a(b)(c)}$ and $\mathtt{b(a(c))}$ represent the identical topology with different roots. Even with the same root, tree string $\mathtt{a(b)(\ c)}$ and $\mathtt{a(c)(b)}$ represent the same tree. Since any node (capable of bus management) on a FireWire bus can be the root, we 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 (Canonicalizer H) A Canonicalizer 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: $\forall$**u** in t, **u** and its all children are sorted in lexicographical order. Tree string t_1=$\mathtt{a(b\ (c)(d))(e(f\ ))}$ is in its canonical form. Tree string t_2=$\mathtt{a(e(f))\ (b(c)(d))}$ is not since **a** and its children **e** and **b** are not sorted. Hence we have $t_1 = H(t_2)$.

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

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

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

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

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

We are interested in a specific set T of tree strings: each tree string in T contains all nodes in U; all the tree strings have the same root node; no two tree strings in T have the same canonical representations. T represents a complete set of all the different tree topologies for the node set U.

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

Due to the paper length limitation, the proof of Lemma 1 is omitted. Please refer to [3] for details.

Each node has a limited number of ports available. The root node **d** can have up to $p_{\mathbf{d}}$ children, whereas a non-root node **u** can have up to $p_{\mathbf{u}} - 1$ children (one link to the parent) where $p_{\mathbf{x}}$ is the port count for the node **x**. This is the port count constraint. A tree is *legal* if every node satisfies the port count constraint.

B. Cost Function

Given a transaction $\tau = (u_\tau, v_\tau, s_\tau, w_\tau)$, all the nodes $u \in U$ are categorized into three sets: M_t, M_r, and M_i. $M_t = \{u_\tau\} \cup \{v_\tau\}$ consists of two communicating nodes, where u_τ and v_τ are the sender and the receiver, respectively. 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 power mode m_u of the node u in each above sets are transferring, repeating, and idle, respectively.

For a given node u, the power function P is a function of the port count p_u and mode m_u, denoted as $P(p_u, m_u)$. The power function can be a lookup table whose data entries come from manufacturer's data sheets [4].

We define the power function of a tree t as:

$$P(\tau,t) = \sum_{u \in M_t} P(p_u,m_u) + \sum_{u \in M_r} P(p_u,m_u) + \sum_{u \in M_i} P(p_u,m_u) \quad (3)$$

The power function $P(\tau,t)$ represents the total bus power of the network during the transaction τ, including power of transaction nodes (both transferring and repeating) and idle nodes.

For a transaction τ, the *effective transaction time* (ETT) is defined as $D_\tau = w/s$, where w is the workload, and s is the transmission speed. During a given time period D, we suppose there are k transaction instances $\{\tau_i | i = 1,\ldots,k\}$. The total ETT for the transaction τ is: $D_\tau = \sum_i D_{\tau_i}$. We define *utilization* of the transaction τ as: $\lambda_\tau = D_\tau / D$. Finally for a given tree string t, we define our cost function as:

$$C = \sum_{\tau \in \Gamma} P(\tau,t)\lambda_\tau \quad (4)$$

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

Fig. 3. The tree enumeration algorithm.

C. Problem Statement

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

$$\forall\ \tau(u_\tau, v_\tau, s_\tau, w_\tau) \in \Gamma \text{ and } \forall\ x \in M_r,\ s_\tau \in S_x. \qquad (5)$$

That is, for a transaction, all the intermediate nodes on a routing path should support the transfer speed. We aim to find trees that have the minimum cost defined by (4). The input to the problem is the node set U and the transaction set Γ. The output of the problem is a tree (or a set of trees) with the minimum cost.

We add *hubs* to the node set in case it cannot form a single tree (i.e., the total port count is less than $2|U|-2$). A hub repeats transactions but cannot be a peer node. Several types of hubs are available, differentiated by their port counts and power consumption. We are interested in finding out which hub type is energy-optimal in connecting the node devices.

III. Algorithm

Our algorithm (shown in Fig. 3) incrementally generates tree topologies using the grammar-based growing function F. We use F to add a node to an existing tree either as a leaf node or as an internal node. At each step, a tree topology is dropped if it fails to satisfy constraints. The while loop (line 7–14) generates new trees and expands the tree set. Two main steps, AddAsLeaf() and AddAsBranch(), add a new node to the existing tree as a leaf node and as an internal node, respectively. Fig. 4 shows the procedure AddAsLeaf(). When adding a new node x to an existing tree t as a leaf node, we attach x to each node if it has an available port. We identify the routing path between two communicating nodes, and check speed constraints. If the tree satisfies speed constraints for all transactions, we append it to the tree set. The other step AddAsBranch() (not shown) to add x as an internal node is

```
AddAsLeaf($t$, $x$, $\Theta$)
1   # Input : tree $t$, node $x$, transaction set $\Gamma$
2   # Output: tree set $T_l$
3   $T_l \leftarrow \emptyset$; $ptr \leftarrow 0$
4   while $ptr < len(t)$
5       while $t[ptr] \notin U$ { $ptr \leftarrow ptr+1$} # Find next node id
6       if $p[t[ptr]] > 0$ # If port available
7           $T_{sub} \leftarrow Subtree(t[ptr])$ # $T_{sub}$: a set of subtrees of $t[ptr]$
8           insertx$(T_{sub},'(x)')$ # Keep elements in $T_{sub}$ sorted
9           $t' \leftarrow$ join$(T_{sub})$ #Concatenate elements in $T_{sub}$ into a string
10          $t'' \leftarrow$ insertSub$(t,t')$ # Substitute $t[ptr]$'s subtrees for $t'$
11          updatePort$(p)$ # Update port count information
12          $tag \leftarrow 1$
13          for each $\tau$ in $\Gamma$
14              if checkSpeed$(t'',\tau)$ == FALSE
15                  $tag \leftarrow 0$; break
16          if $tag$ == 1 { $T_l \leftarrow T_l \cup \{t''\}$ }
17      $ptr \leftarrow ptr+1$
18  return $T_l$
```

Fig. 4. The AddAsLeaf routine.

similarly implemented as string manipulation. A global search procedure enumerates all feasible trees, calculates the cost for every tree, and finds tree(s) with the minimum cost.

Here we briefly discuss the algorithm complexity. An exhaustive approach will generate $n!2^{n-1}$ trees for n nodes (see [3] for details), some of which are either redundant or infeasible. On the other hand, our algorithm generates tree strings in their canonical form only. In AddAsLeaf(), the if-branch (line 6–16) produces at most k strings (k is the number of nodes). The AddAsBranch() routine produces at most $(k-1)$ strings. Thus we have at most $(2k-1)$ tree strings for a tree of $(k+1)$ nodes. Theoretically, our algorithm produces at most $\prod_{i=0}^{n-2}(2n-3-2i)$ strings for n nodes. It is already asymptotically smaller than the exhaustive approach. In reality, our algorithm generates much fewer trees since we apply constraints in each step, significantly reducing the number of trees generated in that step and in the following steps.

IV. Experimental Results

We apply our algorithm to two FireWire bus examples. We use Firebug [2], a software bus snooping tool, to monitor the bus traffic and obtain the workload information. Our algorithm generates optimal tree sets efficiently. Potential energy saving is achieved by choose the trees with the minimum cost.

Example I

We have eight devices (two Mac computers, a PC, a hard drive, a camcorder, two web cameras, and a hub) connected with FireWire bus interfaces, as listed in Table I(a). We first arbitrarily interconnect all the devices and turn on FireBug to

Device	s(Mb/s)	p
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

(a)

τ	u1	u2	s(Mb/s)	w(Gb)
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

(b)

TABLE I
(A) A LIST OF FIREWIRE DEVICES; (B) A LIST OF TRANSACTIONS

	Example I				Example II		
Hub type	$p=3$	$p=4$	$p=6$		$p=3$	$p=4$	$p=6$
# of nodes	8	8	8		13	12	11
# of trees	90	269	376		45761	17001	2013
MaxCost	270.6	306.2	338.9		332.8	304.4	270.4
MinCost	243.2	267.5	290.8		300.9	268.8	236.7
diff(%)	12.2	14.5	16.6		10.1	13.3	14.2
# of solutions	4	1	1		3	2	1

TABLE II
EXPERIMENT RESULTS: EIGHT NODES FOR EXAMPLE I AND ELEVEN OR MORE NODES FOR EXAMPLE II.

monitor the traffic on the bus, and then extract transaction-related information, and obtain the transaction table shown in Table I(b).

Table II shows the experimental results. Exhaustive enumeration will produce $5,160,960$ trees (see the previous section), while our algorithm shrinks the tree set sizes down to 90–376. *MaxCost* and *MinCost* are the maximum and minimum cost value for all feasible trees. In three cases ($f_n = 3,4,6$), the differences between *MinCost* and *MaxCost* range from 12.2% to 16.6%, representing the potential energy savings by selecting the trees with *MinCost*.

Note 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. Four trees with the minimum cost are found when using a three-port hub (see Fig. 5).

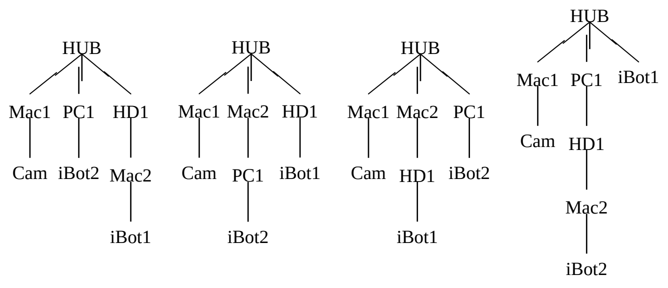

Fig. 5. Example I: four optimal trees found.

Example II

We use three Mac computers, four FireWire hard drives, a printer, a scanner and a camcorder, totaling ten devices. To satisfy the connectivity condition, we add three, two, and one hub when using three-port, four-port, and six-port hubs, respectively. For the exhaustive approach, the problem of up to thirteen nodes becomes intractable in practice. Our algorithm generates highly compact tree sets. Potential energy savings range from 10.1% to 14.2%.

Note that we only consider the time periods with traffic on the bus. When the bus is completely idle, some or all of the bus nodes can be disabled. In the implementation of FireWire bus drivers, the link layer and above layers can be disabled for power reduction. In our examples, we assume all the layers are on all the time. Even for the physical layer controllers, dynamic power management techniques can be applied to disable them when there is no traffic passing through them. The above conditions are orthogonal to our techniques. This means that additive energy saving could be achieved by combining our technique together with other power management techniques.

V. CONCLUSION

This paper presents a method for optimizing peer-to-peer bus topology for energy reduction. We represent trees with a canonical string form, which is both concise and easy to manipulate. We purpose an incremental approach to enumerate tree topologies. By applying a number of constraints to tree growing steps, we are able to obtain both compact and complete tree sets without producing redundant trees. We capture the bus workload information by monitoring the bus traffic and factor it into the cost function. The 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 configuring it to form logic trees for performance, energy-saving, and fault-tolerance reasons.

REFERENCES

[1] D. Anderson. *FireWire System Architecture*. MindShare Inc., Reading, Massachusetts, second edition, 1999.

[2] Apple Inc. Apple's FireWire SDK 2.8.1. In *ftp://ftp.apple.com/developer/Development_Kits/*, 2000.

[3] D. Li. Topology selection for energy minimization in embedded networks. In *Technical report, IMPACCT-TR-09-01-02, University of California at Irvine, http://www.ece.uci.edu/~dli/research02/impacct-tr-09-01-02.pdf*, September 2002.

[4] Texas Instruments. IEEE 1394 products: Integrated devices, link layer controllers and physical layer controllers. In *http://www.ti.com/sc/1394*, 2002.