The Energy and Entropy of VLSI Computations*

José A. Tierno†, Rajit Manohar, and Alain J. Martin
Department of Computer Science
California Institute of Technology
Pasadena, CA 91125. USA.

Abstract

We introduce the concept of energy index, a measure which can be used to estimate the power dissipation of a standard implementation of the high-level specification for an asynchronous circuit. This energy index is related to information-theoretic entropy measures. It is shown how these measures can be used to design low-power circuits.

1 Introduction

In CMOS technology, energy is dissipated only when a node of the circuit is switched—when a capacitor is charged or discharged.

Therefore, an energy model for VLSI computation based on CMOS may assume that energy is dissipated only when the state of the computation changes—the process of waiting does not dissipate any power. This assumption is satisfied by a CMOS asynchronous circuit, but is not in general satisfied by a clocked circuit. This difference accounts for the important advantage of asynchronous design over synchronous clocked design [6, 8, 11] as far as power is concerned.

According to this model, energy-efficient computations are those in which work is performed only when necessary. But, then, how can we define and compute the minimal amount of work necessary to carry out a given computation? Can we achieve this lower bound in an actual implementation?

Can we use knowledge about the lower bound to direct the synthesis of an (asynchronous) circuit implementing the computation? This paper proposes an answer to these questions by introducing a model that can predict the energy required for the execution of a given sequence of actions.

We base the energy model on the CSP [3] specification of an asynchronous circuit. The CSP description for an asynchronous circuit is general, and can be synthesized directly into a transistor network. Using what we know about the synthesis procedure, we can predict the energy cost of CSP constructs, in particular the cost of synchronization. This model can be easily translated to other specification languages.

*The research described in this report was sponsored by the Advanced Research Projects Agency and monitored by the Office of Army Research.

†Now at IBM T.J. Watson Research Center, Yorktown Heights, NY 10598.

The set of all possible sequences of input/output actions is used as an implementation-independent specification of an asynchronous circuit. We assume that the circuit is finite and deterministic.

This paper is organized as follows: In Section 2, we define an energy model based on the energy cost of implementing synchronization in asynchronous circuits derived from a CSP specification. In Section 3, we relate this energy model to the information-theoretic entropy of the sequences of input/output actions the circuit has to execute, and derive a lower bound to the energy cost of any circuit that has to perform the same computation. In Section 4 we use trace theory to derive similar results to section 3 in a way that is independent of the specification language. Finally, in Section 5 we apply these results to show how the synthesis procedure can be directed so that we obtain an energy-optimal circuit.

2 Energy Model for CSP Processes

In this section, we show how we can derive a simple energy model from the CSP specification of a CMOS asynchronous digital circuit. The CSP specification of an asynchronous circuit corresponds very closely to its implementation: for each assignment, communication and function evaluation executed by the CSP program there is a corresponding assignment, communication, function evaluation computed by the CMOS implementation. In fact, one can syntactically transform a CSP program into CMOS [1]. The CMOS implementation will dissipate energy only during the execution of various parts of the CSP program; therefore, this energy can be attributed to the energy required to execute the corresponding CSP statement. To calculate the energy required to execute a CSP program, we add the energy required to execute each statement in a "canonical" trace of that program; we can also use the relative frequencies of occurrence of each statement in the program on a reasonably large set of typical traces.

The purpose of the model presented here is to study architectural trade-offs (e.g., comparison of bit-serial and parallel implementations of a function) or to determine architectural parameters (e.g., the optimal width of a cache memory) with respect to energy consumption. A detailed model with a large number of parameters can be intractable without significantly increasing the accuracy of the model, especially if the parameters are layout-dependent (and, therefore, not well known before the layout is complete). A simpler model is desirable at the design stage; we will base this model on the cost of communication, assignment, and selection.

The model proposed is based on the *energy performance index*. To each type of statement, we assign an index that is representative of the energy that we expect that operation to cost in a typical implementation.

2.1 Energy Index

CMOS circuits have three main sources of energy dissipation: leakage currents, short-circuit currents, and dynamic currents. The total energy dissipated during the execution of one operation, E_T, can be calculated as:

$$E_T = E_s + E_d + E_{sc} \tag{1}$$

where E_s is the energy dissipated by the sub-threshold leakage currents, E_d is the energy used for charging and discharging capacitors, and E_{sc} is the energy dissipated by the short-circuit currents.

Leakage currents come from the sub-threshold behavior of MOSFET's, and constitute a small part of the total power dissipation in modern CMOS processes. Short-circuit currents originate in the short transients that occur when both pull-up and pull-down transistors conduct while the input signal switches between V_{thn} and $V_{DD} - V_{thp}$. We will assimilate this switching energy to the dynamic energy dissipation, that represents the bulk of the total energy dissipation in a standard CMOS circuit. Dynamic energy dissipation, E_d, comes from the energy used to charge the capacitors in the circuit. The capacitors are then discharged to ground, and the energy is not recuperated. E_d can be computed as:

$$E_d = \sum_{C_i} n_i C_i V_{DD}^2 \tag{2}$$

where the C_i's are all the capacitors in the circuit, and n_i is the number of times capacitor C_i is switched in the execution of one operation. We rewrite Eq. 2 as:

$$E_d = K_L V_{DD}^2 \tag{3}$$

Based on these results, we use as an energy performance index for an asynchronous CMOS circuit the corresponding constant K_L. This index is independent of the power-supply voltage and the speed of operation; furthermore, K_L is additive: we can calculate the index corresponding to an operation by adding the indices of all of its sub-operations.

2.2 Synchronization

We present next the part of the energy model that relates to the cost of synchronization between CSP statements. The cost of data operations is explained in [10].

The synchronization primitives of CSP are parallel composition ('$\|$'), sequential composition (';'), guarded choice ('$\square$'), repetition, and bullet synchronization between communication actions ('$\bullet$'). Some of these primitives have zero energy cost, such as parallel composition. Some of these primitives require extra hardware to be implemented, such as guarded choice.

Concurrency:
A basic postulate of this model is that parallel composition is free: no extra circuits are required in the implementation. If there is no synchronization between the P_i processes, we can write:

$$\mathcal{C}(\langle \| i : 1..n : P_i \rangle) = \sum_{i=1}^{n} \mathcal{C}(P_i) \tag{4}$$

where $\mathcal{C}()$ is the cost function that assigns an energy index to a program.

If processes synchronize, we must first determine the relative frequencies of their execution, and consider the weighted sum of their individual costs. Given these weights w_i, we have:

$$\mathcal{C}(\langle \| i : 1..n : P_i \rangle) = \sum_{i=1}^{n} w_i \mathcal{C}(P_i) \tag{5}$$

Example: Consider the concurrent composition of the following CSP processes:

$P \equiv \ *[L?x;\ \ R!x]$
$Q \equiv \ *[R?x; R1!x; R?x; R2!x]$

Every execution of Q corresponds to two executions of P. Therefore, when computing the energy cost of $P \| Q$, we weight P by 1 and Q by $\frac{1}{2}$. This cost corresponds to the cost of an execution when an L action is executed once.

Sequencing:
Sequencing synchronizes the end of an action with the beginning of the next action. If the previous action is the parallel composition of several actions, the end of those actions has to be synchronized with a tree, which has a linear energy cost. If the next action is the parallel composition of several actions, the start signal has to be distributed to them (maybe with a tree), which also has a linear cost (see Fig. 1). We can express these costs with the following equation:

$$\begin{aligned}
&\mathcal{C}(\langle \| i : 1..n : P_i \rangle ; \langle \| j : 1..m : Q_j \rangle) = \\
&\qquad \mathcal{C}(\text{join}(n)) + \mathcal{C}(\text{fork}(m)) + \mathcal{C}(;) + \\
&\qquad \mathcal{C}(\langle \| i : 1..n : P_i \rangle) + \mathcal{C}(\langle \| j : 1..m : Q_j \rangle) \\
&= \quad K_j(n-1) + K_f(m-1) + K_{sc} + \\
&\qquad \sum_{i=1}^{n} \mathcal{C}(P_i) + \sum_{j=1}^{m} \mathcal{C}(Q_j)
\end{aligned} \tag{6}$$

where the P_i and Q_i processes have no synchronizations between them, and the K_j, K_f, and K_{sc} are technology-dependent constants. If $n = 1$ and $m = 1$, then the fork and join circuits are not needed, and the cost of sequencing is just the constant overhead K_{sc}.

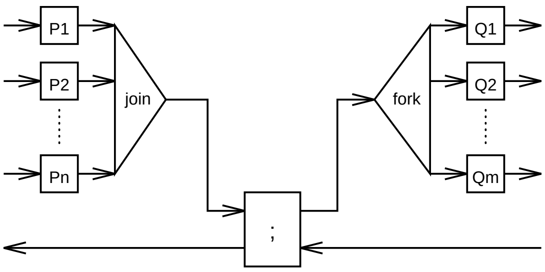

Figure 1: Extra circuits required to implement sequencing between two blocks of concurrent processes. The *fork* and *join* trees can be configured in several ways.

Choice:

Guarded choice can be implemented in a number of ways. We consider the cost of selection as the difference in cost between the following two programs:

$$PAR \equiv \langle \ \| \ i : 1..n : \ \ *[[\ G_i \longrightarrow A_i \ ]] \rangle$$

and,

$$CHOOSE \equiv *[[\langle \ \Box \ i : 1..n : \ \ G_i \longrightarrow A_i \ \rangle]]$$

Program *CHOOSE* can be transformed into program *PAR* using P and V operations on a semaphore:

$$\begin{array}{l} CHOOSE \equiv \\ \quad \langle \| \, i : 1..n : *[[\ G_i \longrightarrow P; \ \ [G_i \longrightarrow A_i \\ \qquad\qquad\qquad\qquad\qquad\qquad \Box \neg G_i \longrightarrow \mathbf{skip} \\ \qquad\qquad\qquad\qquad\qquad\qquad]; V \ ]] \rangle \end{array}$$

The cost of choice is, therefore, the cost of implementing that semaphore. A number of implementations are possible and practical; the semaphore may be implemented with a selection tree:

$$\begin{array}{l} CHOOSE \equiv \\ \quad \langle \ \| \ i : 1..N : \ \ *[[\ G_i \longrightarrow U_i; A_i; U_i \ ]] \rangle \\ \\ \quad \| \ *[[\langle \ \Box \ \ i : 1..N/2 : \ \ \overline{U}_i \longrightarrow L; U_i; U_i; L \ \ \rangle]] \\ \\ \quad \| \ *[[\langle \ \Box \ \ i : N/2+1..N : \\ \qquad\qquad \overline{U}_i \longrightarrow H; U_i; U_i; H \ \ \rangle]] \\ \\ \quad \| \ *[[\ \overline{L} \longrightarrow L; L \ \ \Box \ \ \overline{H} \longrightarrow H; H \ \ ]] \end{array}$$

We apply this transformation recursively, and we get a cost of selection that is logarithmic in the number of choices and can be expressed as:

$$\mathcal{C}([\langle \Box \ i : 1..n : G_i \rightarrow A_i \rangle]) = \\ K_c \log_2 n + \sum_{i=1}^{n} p_i \mathcal{C}([G_i]; U_i; A_i; U_i) \qquad (7)$$

where p_i is the probability of picking guard G_i.

Arbitration does not affect the energy cost, since two-way arbiters only have a constant energy cost on average. To show this, observe that the probability of a two-way arbiter taking time t to arbitrate is proportional to e^{-kt} for some constant k. Assuming that the energy dissipated is proportional to time, the average energy dissipation is $\int_0^\infty Ate^{-kt}\,dt = A/k^2$ for some constant A. Since n-way arbiters can be constructed by using a tree of two-way arbiters, the total energy cost of an arbitrated choice is also logarithmic in the number of alternatives.

3 Energy and Entropy

In this section we introduce a constrained version of the CSP language, show how we can translate any CSP process into this constrained CSP. We then define an energy model for the constrained representation of a process, and derive a lower bound to this energy model based on the information-theoretic entropy of the input/output behavior of the specification of the CSP process. This lower bound does not depend on the particular implementation of the process, only on its expected input/output behavior, and therefore applies to any circuit that implements that behavior. The energy model measures the number of "elementary" transitions executed by the circuit assuming a CMOS-style digital circuit.

3.1 Flat CSP

A CSP process P is called *flat* if it has the form:

$$\begin{array}{l} P \equiv *[[\ G_0 \longrightarrow \ \ A_0; \ \ SA_0 \\ \qquad\ \Box \ G_1 \longrightarrow \ \ A_1; \ \ SA_1 \\ \qquad\ \ldots \\ \qquad\ \Box \ G_{n-1} \longrightarrow \ \ A_{n-1}; \ \ SA_{n-1} \\ \quad]] \end{array}$$

where the guards G_i are stable (that is, once they become true, they remain true at least until the first action of the guarded command is executed). Commands A_i are either data-less or boolean communications, and statements SA_i are constant assignments to state variables. This restriction is introduced so that all the complexity of the computation is handled by the selection mechanism, instead of the data-assignment mechanism.

A CSP process can be transformed into flat form by applying the following rules recursively. The variables that are added when applying a rule are new; that is, they did not appear in the original program and are initially false. S_A represents a sequence of state variable assignments. The notation $A \triangleright B$ is used to give a transformation from program A to program B.

Sequencing. This transformation removes sequential composition. Other state assignments are possible to enforce sequencing.

$$\Box \ G \longrightarrow \ \ A_0; A_1; \ldots; A_n; \ \ S_A \ \triangleright$$

$$\begin{array}{l} \Box \ \neg s_G \wedge G \longrightarrow \ \ s_{A_0}\uparrow; s_G\uparrow \\ \Box \ s_G \wedge s_{A_0} \longrightarrow \ \ A_0; s_{A_0}\downarrow; s_{A_1}\uparrow \\ \ldots \\ \Box \ s_G \wedge s_{A_n} \longrightarrow \ \ A_n; s_{A_n}\downarrow; s_G\downarrow; S_A \end{array}$$

Choice. This transformation removes choice composition.

$$
\begin{array}{l}
\square\ G \longrightarrow\ [G_0 \longrightarrow A_0 \square \ldots \square G_n \longrightarrow A_n];\ S_A\ \triangleright \\
\quad \square\ \neg s_G \wedge G \longrightarrow\ s_G\uparrow \\
\quad \square\ s_G \wedge G_0 \longrightarrow\ A_0;\ s_G\downarrow; S_A \\
\quad \ldots \\
\quad \square\ s_G \wedge G_n \longrightarrow\ A_n;\ s_G\downarrow; S_A
\end{array}
$$

Repetition. This transformation factors out repetition.

$$
\begin{array}{l}
\square\ G \longrightarrow\ *[G_0 \longrightarrow A_0 \square \ldots \square G_n \longrightarrow A_n];\ S_A\ \triangleright \\
\quad \square\ \neg s_G \wedge G \longrightarrow\ s_G\uparrow \\
\quad \square\ s_G \wedge G_0 \longrightarrow\ A_0 \\
\quad \ldots \\
\quad \square\ s_G \wedge G_n \longrightarrow\ A_n \\
\quad \square\ s_G \wedge \neg G_0 \wedge \ldots \wedge \neg G_n \longrightarrow\ s_G\downarrow;\ S_A
\end{array}
$$

3.2 Flat Process Decomposition

A single flat process is an inefficient implementation of a CSP program; all the guards have to be evaluated every time a statement is executed. Hierarchical evaluation of guards can improve the average cost per statement by making the statements most frequently executed cheaper at the expense of the more infrequent ones.

Hierarchical decomposition transforms one flat process into two, in the following way:

$$
\begin{array}{l}
*[[G_0 \longrightarrow A_0; SA_0 \\
\ \square G_1 \longrightarrow A_1; SA_1 \\
\ \ldots \\
\ \square G_{n-1} \longrightarrow A_{n-1}; SA_{n-1} \\
\ ]] \\
\triangleright \\
\quad *[[\ G_0 \vee G_1 \vee \ldots \vee G_{j-1} \longrightarrow\ H \\
\qquad \square\ G_j \longrightarrow\ A_j;\ SA_j \\
\qquad \ldots \\
\qquad \square\ G_{n-1} \longrightarrow\ A_{n-1};\ SA_{n-1} \\
\quad]] \\
\| *[[\ \overline{H} \wedge \neg G_0 \wedge \neg G_1 \wedge \ldots \wedge \neg G_{j-1} \longrightarrow\ H \\
\qquad \square\ \overline{H} \wedge G_0 \longrightarrow\ A_0;\ SA_0 \\
\qquad \ldots \\
\qquad \square\ \overline{H} \wedge G_{j-1} \longrightarrow\ A_{j-1};\ SA_{j-1} \\
\quad]]
\end{array}
$$

where H is a new channel.

The two resulting processes are also flat, and we can re-apply the procedure to each of them to obtain a tree of flat subprocesses.

3.3 Entropy

We define the energy complexity of the hierarchical decomposition of a flat process and relate this energy complexity to the information-theoretic entropy of the sequences of input/output symbols.

Hierarchical decomposition of a flat process can drastically reduce the average energy cost of executing a CSP process. Two mechanisms are at play: first, we can choose the decomposition so that the more frequently executed statements are "higher up" in the decomposition tree, and, second, the state of the tree of processes stores information about the history of the computation, modifying the cost of each statement according to this history.

Given a hierarchical decomposition of a flat process, an execution of that process corresponds to a path in the tree of subprocesses. This path can be encoded by giving the sequence of cardinals of the guarded command selected within each subprocess. Given this sequence and the program text, we can reconstruct the computation.

Example: Consider the program

$$
\begin{array}{l}
P1 \equiv \\
\quad *[[\ G_1 \longrightarrow H_1\ \square\ G_2 \longrightarrow A_1\ \square\ G_3 \longrightarrow A_2\ ]] \\
P2 \equiv \\
\quad *[[\ \overline{H}_1 \wedge G_4 \longrightarrow\ A_3\ \square\ \overline{H}_1 \wedge G_5 \longrightarrow\ H_1\ ]] \\
P \equiv\ P1\ \|\ P2
\end{array}
$$

where the A_i's are input/output symbols. The execution begins in process $P1$ (since $\overline{H_1}$ is false). The execution of A_3, A_1, A_2 corresponds to the following sequence of selections: first guard in $P1$; first guard in $P2$; second guard in $P2$; second guard in P1; third guard in $P1$. Therefore, the cardinal sequence 1, 1, 2, 2, 3 corresponds to the sequence A_3, A_1, A_2; similarly, the cardinal sequence 1, 2, 2, 2, 3 corresponds to the sequence A_1, A_1, A_2.

We have already shown that the cost of executing a guarded command from a one-of-n selection scales with $\log_2 n$. To simplify the notation and the proofs in this section, we use $\lceil \log_2 n \rceil$[1] as the energy complexity of one-of-n selection.

To formalize, let P be a process, FP be the flat representation of that process, HP be a hierarchical decomposition of FP, and $a_{1..m} = a_1, \ldots, a_m$ be the first m statements executed by process P. Let $s_1, \ldots, s_{k(\mathrm{HP}, a_{1..m})}$ be the sequence of cardinals of the selected guarded commands required to reconstruct $a_{1..m}$ from HP, and $k(\mathrm{HP}, a_{1..m})$ be the length of that sequence. This sequence can be encoded as a list of $l(\mathrm{HP}, a_{1..m})$ bits, $K_1, \ldots, K_{l(\mathrm{HP}, a_{1..m})}$. Finally, let $\mathcal{C}(\mathrm{HP}, a_{1..m})$ be the cost of running process HP until $a_{1..m}$ has been executed.

We can relate the energy cost to the length of the code with the following theorem:

Theorem 1 $\mathcal{C}(\mathrm{HP}, a_{1..m}) = l(\mathrm{HP}, a_{1..m})$.

Proof: Given n_i, the number of guarded commands in the i^{th} selection and $\lceil \log_2 n_i \rceil$ the cost of that selection, we have:

$$
\mathcal{C}(\mathrm{HP}, a_{1..m}) = \sum_{i=1}^{k(\mathrm{HP}, a_{1..m})} \lceil \log_2 n_i \rceil
$$

We need $\lceil \log_2 n_i \rceil$ bits to encode the i^{th} cardinal;

[1] $\lceil x \rceil$ represents the ceiling function of x, the smallest integer that is at least x.

therefore,

$$l(\mathrm{HP}, a_{1..m}) = \sum_{i=1}^{k(\mathrm{HP}, a_{1..m})} \lceil \log_2 n_i \rceil = \mathcal{C}(\mathrm{HP}, a_{1..m})$$

∎

From Theorem 1 we conclude that optimizing the energy cost of a CSP process requires finding the HP that best encodes the statement sequence $a_1, \ldots, a_m$. First we compute a lower bound of the optimum encoding, using some results from information theory.

Let A_i be a random variable that takes as value the i^{th} statement executed by process P. The statement sequences of length m have a probability distribution $\Pr(A_{1..m})$, which can be calculated either deterministically (for example, assuming that all input sequences are possible and equiprobable), or statistically, by looking at actual traces of the execution of the program. Let S_m be the set of all statement sequences with non-zero probability.

To define the cost per statement, we take an average of the energy cost of the process over a very large number of statements. The limit of the average cost when the number of statements goes to infinity may not exist, or be unbounded (as would be the case in a busy-waiting loop). To avoid those problems, we use the lim sup [2] in the following definition:

Definition 1 *The cost per statement of a process* HP, $\mathcal{C}(\mathrm{HP})$*, is defined as:*

$$\mathcal{C}(\mathrm{HP}) = \limsup_{m \to +\infty} \sum_{(a_{1..m}) \in S_m} \Pr(a_1 \ldots a_m) \frac{1}{m} \mathcal{C}(\mathrm{HP}, a_{1..m}) \tag{8}$$

The following theorem gives a sufficient, though not necessary, condition for the convergence of this limit:

Theorem 2 *If every loop of HP (that is, every sequence of statements from HP that has the same initial state and final state) contains at least one statement from P, then* $\mathcal{C}(\mathrm{HP})$ *converges.*

Proof: Let K be the length of the longest loop, counted as the number of selections made in that loop, and n be the number of statements in P. Then we can write $k(\mathrm{HP}, a_{1..m}) \leq K \times m$; therefore, $\mathcal{C}(\mathrm{HP}, a_{1..m})$ is bounded above by $K\lceil \log_2 n \rceil$, and the lim sup converges. Details of this proof can be found in [10]. ∎

The entropy of the statement sequences of length m, $H(A_1, \ldots, A_m)$, is defined in the usual way [9]:

[2]
$$\limsup_{n \to +\infty} a_n = \lim_{n \to +\infty} \sup_{i > n}(a_i)$$

Definition 2 *Let* $(A_1, \ldots, A_m)$ *be a sequence of random variables. The entropy of this sequence,* $H(A_1, \ldots, A_m)$*, is defined as:*

$$H(A_1 \ldots A_m) = \sum_{(a_{1..m}) \in S_m} \Pr(a_1 \ldots a_m) \log_2 \frac{1}{\Pr(a_1 \ldots a_m)} \tag{9}$$

We use the following theorem to define the entropy of a process P:

Theorem 3 *The limit,*

$$\limsup_{m \to +\infty} \frac{1}{m} H(A_1, \ldots, A_m)$$

always exists.

Proof: If A_i can take n different values, we have

$$\begin{aligned} 0 &\leq \frac{1}{m} H(A_1 \ldots A_m) \\ &\leq \frac{1}{m} \left(H(A_1) + \cdots + H(A_m)\right) \\ &\leq \log_2 n \end{aligned}$$

The sequence is bounded by a constant; therefore, the lim sup exists. ∎

Definition 3 *The entropy of a process* P*,* $H(\mathrm{P})$*, is defined as:*

$$H(\mathrm{P}) = \limsup_{m \to +\infty} \frac{1}{m} H(A_1, \ldots, A_m) \tag{10}$$

Now we are ready to prove the basic theorem that gives a lower bound to the energy complexity of a hierarchical decomposition of a process P:

Theorem 4 *For every process* P*, and every hierarchical decomposition* HP *of* P*, we have:*

$$H(\mathrm{P}) \leq \mathcal{C}(\mathrm{HP}) \tag{11}$$

Proof: $K_1, \ldots, K_{l(\mathrm{HP},m)}$ is a prefix-code[3] for $A_1, \ldots, A_m$; therefore we know that the average length of the code is at least the entropy of the source of symbols [9], and we can write:

$$\begin{aligned} H(A_1, \ldots, A_m) &\leq \sum \Pr(a_1, \ldots, a_m) l(\mathrm{HP}, a_{1..m}) \\ &= \sum \Pr(a_1, \ldots, a_m) \mathcal{C}(\mathrm{HP}, a_{1..m}) \end{aligned}$$

Dividing by m and taking lim sup on both sides of the inequality, we get the thesis. ∎

Theorem 4 gives a lower bound to the energy cost of a hierarchical decomposition. The next question to be answered is under what conditions the lower bound can be reached. The following theorem gives a partial answer:

[3] A prefix-code is a code such that no codeword is a prefix of another codeword.

Theorem 5 *If for every sequence of statements* $a_1, \ldots, a_i$ *executed by a hierarchical decomposition* HP *of a process* P *and for a constant* K, *the following conditions hold:*

1. $\Pr(s_{k(\mathrm{HP},a_{1..i})} | s_1, \ldots, s_{k(\mathrm{HP},a_{1..i})-1}) = \frac{1}{n_{k(\mathrm{HP},a_{1..i})}}$
2. $\Pr(s_1, \ldots, s_{k(\mathrm{HP},a_{1..i})}) = \Pr(a_1, \ldots, a_i)$
3. $k(\mathrm{HP}, a_{1..i}) < K \times i$,

then $H(\mathrm{P}) \le \mathcal{C}(\mathrm{HP}) \le H(\mathrm{P}) + K$ *holds.*

Theorem 5 can be interpreted as follows. The first condition means that all choices in a computation of HP are equally probable. The second condition is automatically verified if, for each sequence of statements from the original process P, there is a unique sequence of choices from HP. The third condition puts a fixed bound to the overhead introduced by the hierarchical decomposition. It is satisfied if each loop contains at least one statement from P. This condition excludes busy-waiting.

Proof: The proof of this theorem can be found in [10]. ∎

The entropy $H(\mathrm{P})$ of a program P was defined based on the entropy of the sequences of statements from the original program P. We can restrict this definition to the input symbols or the output symbols, exclusively, and all the theorems proved so far hold as well.

4 Trace Entropy

So far we have related the energy index of a CSP program to the energy dissipated by a standard VLSI implementation of the program. The energy index of a flat CSP program was shown to be related to the average length of an execution sequence.

We now propose an alternative method for computing the entropy of a program. The approach is based on trace theory [2, 12]. Traces can be used to describe asynchronous circuits that are designed by various methods. A set of traces can be considered to be the specification of a circuit. Therefore, entropy measures on trace sets are not dependent on any particular design technique. The measure is related to the *specification* rather than the implementation of the circuit.

4.1 Closed Programs

A circuit is said to be *closed* if it does not have any dangling inputs. In a closed VLSI computation, all possible behaviors of the circuit are known apriori. As a result, a computation can be considered to be a set of traces, where a trace consists of a sequence of state transitions. We assume that assignment to boolean variables is the only atomic action. Therefore, a state transition can be described in terms of the transition of a single boolean variable. The transition of boolean variable x can be described by the assignment $x := \neg x$, which we abbreviate by x. As a result, a trace structure [12]—a pair (T, A), where T is a set of traces and A is an alphabet—can be used to describe the computation.

An infinite computation dissipates infinite energy. An interesting measure of energy is the amount of energy dissipated per action executed by the computation. To this end, given a trace structure that specifies a computation, we examine the entropy per symbol of that trace structure. This entropy is the average amount of information necessary (per symbol) to specify the computation. However, a trace set specifies more than just a single computation. It specifies a set of possible computations that implement the specification. Therefore, a valid implementation of a trace set would be any nonempty subset of the trace set.

A trace set specifies a subset of the set of all traces of an alphabet. We can specify an implementation of the trace set by specifying a single trace from the set. The amount of information required to specify more than one element from a set is at least as much as that required to specify a single element from it. Therefore, to obtain the minimum amount of information necessary, we need to compute the average information necessary to pick a single trace from a trace set. Abstracting away from traces and trace sets, we can reformulate this problem as follows: how much information is necessary to specify a single element from a subset of a set? To compute this quantity, we assume that the set of possible traces—the specification—is chosen using a uniform distribution.

Theorem 6 *Let* S *be a nonempty subset of* U *of cardinality* N/α, *where* N *is the cardinality of* U. *Let* H_N *be the amount of information necessary to pick a single element from* U *that is contained in* S. *Then,*

$$\lim_{N \to \infty} H_N = \alpha \mathcal{H}\left(\frac{1}{\alpha}\right)$$

where $\mathcal{H}(p) = p \log_2 \frac{1}{p} + (1-p) \log_2 \frac{1}{1-p}$.

Proof: The proof of this theorem can be found in [5]. ∎

Notice that the amount of information necessary to specify an element from an infinite trace set is constant, if the cardinality of the set is a fixed fraction of the set of all traces.

A VLSI circuit is a finite component, and as such, can only have finite history. In fact a VLSI circuit can be modeled as a repetitive computation, in which each iteration has no information about the previous one. Let us examine Theorem 6 in the context of this observation. Let T be the set of all possible traces after one iteration of the computation. For simplicity, we assume that all the traces are of the same length. (If not, one can consider a larger number of iterations.) Let α_0 be the ratio of the cardinality of the set of all possible traces (which have the same length as those in T) to the cardinality of T. If we consider two iterations of the execution, the ratio of cardinality of the two sets of interest will be α_0^2. In general, the ratio will be α_0^N after N iterations.

Lemma 7 *Let $\alpha_0 > 1$ be a fixed constant. Then,*

$$\lim_{N\to\infty} \frac{1}{N} \alpha_0^N \mathcal{H}\left(\frac{1}{\alpha_0^N}\right) = \log_2 \alpha_0$$

Proof: Simplifying $\alpha_0^N \mathcal{H}(\frac{1}{\alpha_0^N})$, we get $\log_2 \alpha_0^N + (\alpha_0^N - 1)\log_2 \frac{\alpha_0^N}{\alpha_0^N - 1}$. As $N \to \infty$, the second term tends to $\log_2 e$ which tends to zero after division by N. The first term is $N \log_2 \alpha_0$, which when divided by N yields the necessary limit. ∎

Therefore, the average energy per iteration depends on the fraction of the traces that describe one iteration of the computation. We can interpret this as follows. As the number of possible execution sequences decreases, we have to restrict the behavior of the circuit. In other words, we have to sequence a larger number of transitions. In the extreme case, we have to sequence every transition in the circuit. As a result, we have to dissipate more energy to enforce this sequencing.

A VLSI circuit consists of a number of gates, each controlling the value of a single boolean variable. The entropy measure in Lemma 7 assumes that we can pick a single trace out of a set of possible traces. Consider the model in which each symbol in the alphabet is treated as an independent quantity [7]. This corresponds to the assumption that an individual gate does not store information about execution history. We compute the probability of occurrence for each symbol in the alphabet for one iteration. The entropy of the trace set is now defined to be the traditional entropy measure of the ensemble[4] $(A, \Pr)$, where $\Pr(a)$ is the probability of the occurrence of symbol $a \in A$ in the trace set.

Property 1 *Let $(A, \Pr)$ be an ensemble, and let A^N be the ensemble that consists of N independent, identically distributed occurrences of A. Then,*

$$\frac{1}{N} H(A^N) = H(A)$$

Therefore, we can conclude that the energy per iteration is, on average, $H(A)$. However, this measure of entropy is not very accurate.

Example: Let $T1 = \{a^\infty, b^\infty, c^\infty\}$ and $T2 = \{(abc)^\infty\}$ be two trace sets[5] over the alphabet $\{a, b, c\}$. Now the measure $H(A)$ would indicate that these two sets are equivalent as far as energy is concerned.

This is clearly false, since we could implement $T1$ by a gate that repeatedly executed a. What we have failed to take into account is the fact that certain traces can occur more frequently than others. The probability of traces and probability of individual transitions form a *joint* distribution.

[4] A pair (X, p) where X is a set of events and p is a probability measure on X is known as an *ensemble*.

[5] s^∞ is the trace that consists of infinite repetitions of sequence s.

4.2 Open Programs

Considering closed programs assumes that a designer has full information about the environment. This assumption is not realistic for large designs. To make the entropy calculation more realistic, we must treat computations as open programs—programs in which the environment is not completely specified.

The major distinction between open and closed programs is that the set of traces that describe an open program are all *potential* execution sequences. As a result, one cannot discard any trace from the trace set. We assume that an execution corresponds to repeatedly selecting a trace from a trace set and executing it. The selection is performed once per iteration of the computation.

We assume that associated with a trace structure is a probability distribution—a joint distribution that describes the probability of each trace in the trace set and the probability of each symbol in the alphabet. This probability distribution can be used to compute the entropy of the set of traces in the usual way. We extend the concept of a trace structure to include the probability distribution of the set of all traces.

Definition 4 (Trace Structure) *A trace structure is the tuple $(T, \Pr, A)$, where*

- *T is a set of strings on the alphabet A;*
- *A is a set of symbols denoting state transitions; and*
- *$\Pr$ is a probability measure on T.*

We extend $\Pr$ to a joint distribution on T and A by considering the frequency of occurrence of each symbol in each trace of T.

Given two different traces in T, one must be able to determine which trace is to be executed in a particular instance of the computation. Since the environment selects the trace to be executed, we will assume that, given a choice in execution, there are certain conditions which one can use to select the appropriate trace for execution. These conditions, or *guards*, are predicates on state variables from A.

Definition 5 (Trace Entropy) *The entropy per symbol of trace structure $\mathcal{T} = (T, \Pr, A)$ is defined to be the entropy of the joint distribution $\Pr$, i.e.,*

$$H(\mathcal{T}) = \sum_{\substack{t\in T\\ a\in A}} \frac{1}{|t|} \Pr(at) \log_2 \frac{1}{\Pr(at)}$$

where $|t|$ is the length of the trace. For infinite traces, the quantity is to be computed by considering the lim sup as the length of the trace tends to infinity.

Notice that this definition is equivalent to the entropy of a process $H(P)$, if each trace has exactly one symbol. Since VLSI computations are repetitive, we can simplify the trace structure to only include finite computations. The set of computations will then be repetitions of the finite traces. For simplicity, we assume that the length of each iteration of the computation is the same.

5 Low Power Synthesis

The trace entropy defined in the previous section can be used to guide the design of low-power programs. In this section we relate the entropy of traces to the energy index of CSP programs.

A finite trace structure describing a repetitive computation has a discrete probability measure associated with it. We can think of a CSP program as an encoding technique that is used to generate the set of possible traces. We can construct a CSP program that achieves an average energy cost equal to the entropy of the set of traces by using an optimal coding technique. One such optimal coding technique is known as Huffman coding, which satisfies the following property:

Property 2 *The average codeword length for a Huffman code is the smallest possible amongst all possible uniquely decodable codes.*

The following definition constructs a CSP program from a trace structure which has a cost that is close to the entropy of the trace structure.

Definition 6 *Let $\mathcal{T} = (T, \Pr, A)$ be a trace structure. Then CSP($\mathcal{T}$), the canonical CSP program associated with $\mathcal{T}$ can be recursively defined as follows.*

Let $S = \{\{t\} \mid t \in T\}$. We extend Pr *to sets of traces by the definition* $\Pr(X) = \sum_{t \in X} P(t)$. *Let $CSP(\{t\}) = t$. Repeatedly apply the following steps until $S = \{T\}$.*

1. *Pick $T1$ and $T2$ from S such that they have the lowest probabilities of occurrence, i.e., $T1 = argmin_{T \in S} \Pr(T)$, and $T2 = argmin_{T \in S, T \neq T1} \Pr(T)$.*
2. *Define $CSP(T1 \cup T2)$ to be the program $[G1 \rightarrow CSP(T1) [] G2 \rightarrow CSP(T2)]$, where $G1$ and $G2$ are the guards that can distinguish between traces in $T1$ and $T2$.*
3. *Replace S by $(S - \{T1, T2\}) \cup \{T1 \cup T2\}$.*

Finally, $CSP(\mathcal{T}) = *[CSP(T)]$.

The definition given above constructs a Huffman tree [4] to select a particular trace for execution. We illustrate the definition given above with the help of the following example.

Example: Consider a trace set with symbols $\{a_0(0.1), a_1(0.1), a_2(0.3), a_3(0.5)\}$, with probabilities given in parentheses. The CSP program constructed by applying the steps above is

$$
\begin{array}{l}
*[\,[\overline{a_3} \longrightarrow a_3 \\
\quad [] \overline{a_0} \vee \overline{a_1} \vee \overline{a_2} \longrightarrow [\overline{a_2} \longrightarrow a_2 \\
\qquad\qquad [] \overline{a_0} \vee \overline{a_1} \longrightarrow [\overline{a_0} \longrightarrow a_0 \\
\qquad\qquad\qquad\qquad [] \overline{a_1} \longrightarrow a_1 \\
\qquad\qquad\qquad\qquad] \\
\qquad\qquad] \\
\quad]\,]
\end{array}
$$

The cost $\mathcal{C}$ associated with this program is $0.5(1) + 0.3(1+1) + 0.1(1+1+1) + 0.1(1+1+1) = 1.7$. The entropy $H(\mathcal{T})$ is given by $-0.1 \log_2 0.1 - 0.1 \log_2 0.1 - 0.3 \log_2 0.3 - 0.5 \log_2 0.5 = 1.685$.

Definition 7 *Given a trace structure $\mathcal{T} = (T, \Pr, A)$, the error $\mathcal{E}(\mathcal{T})$ is the difference between the entropy of the ensemble $(T, \Pr)$ and the average length of a Huffman tree that is derived from the ensemble.*

One of the properties of Huffman codes is that the error function $\mathcal{E}$ is the smallest possible, and that $\mathcal{E}(\mathcal{T}) \leq 1$. Therefore, this loss in entropy is entirely a result of the fact that codewords are of integer length. We formalize the connection between the cost and entropy in the following theorem.

Theorem 8 *Let $\mathcal{T} = (T, P, A)$ be a trace structure where each symbol in a single trace is distinct. Let N be the length of the traces in T. Then,*

$$\frac{1}{N}\mathcal{C}(CSP(\mathcal{T})) = H(\mathcal{T}) + 1 + \frac{\mathcal{E}(\mathcal{T})}{N} - \frac{\log_2 N + 1}{N}$$

Proof: The proof of this theorem follows from the definition of $CSP(\mathcal{T})$, and $H(\mathcal{T})$. ∎

Consider the example given above. The correction factor $1 - \frac{\log_2 N+1}{N}$ is zero (since $N = 1$ in this example), and the cost $\mathcal{C}$ is in fact $H(\mathcal{T})$ plus the discretization error.

Note that given a specification, the parameter N is usually fixed. What Theorem 8 tells us is that the measure $H(\mathcal{T})$ is in fact a measure that can be used to *compare* specifications with some accuracy. The specification can then be used to design a program which achieves the cost given above.

Example: Consider the cost of a simple buffer and an alternator. The buffer can be described by the following CSP:

$$*[L?x; R!x]$$

The trace set corresponding to this is $\{L\,R(1)\}$, which has entropy per symbol $\frac{1}{2}(-0.5 \log_2 0.5 - 0.5 \log_2 0.5) = 0.5$. An alternator can be described by the following CSP specification:

$$*[L?x; R1!x; L?x; R2!x]$$

However, the trace $L\,R1\,L\,R2$ contains repeated instances of L. A better CSP description of an alternator is given by

$$
\begin{array}{l}
*[\,[s \longrightarrow L?x; R1!x \\
\quad [] \neg s \longrightarrow L?x; R2!x \\
\quad];\ \ s := \neg s \\
]
\end{array}
$$

This corresponds to the fact that we need an extra state variable, s, to indicate the current "phase" of the alternator. The trace set corresponding to this CSP specification is $\{L\,R1\,s(0.5), L\,R2\,s(0.5)\}$, and satisfies the conditions of Theorem 8. The entropy of this specification (per symbol) can be computed by applying definition 5, and is equal to 0.86. This indicates that an alternator has a higher energy cost than a buffer (per symbol).

The concurrent composition of alternators, alternating merges, and buffers can be used to construct linear and binary tree buffers. It is easy to see (and calculate) that an alternating merge

$$*[L1?x; R!x; L2?x; R!x]$$

has the same cost as an alternator.

Example: A linear buffer stage has a cost of 0.5 per symbol. The total cost of a single buffer stage is $2 \times 0.5 = 1.0$ (there are two symbols executed). The cost of an alternator (and a merge) is 0.86 per symbol, corresponding to a total cost of $3 \times 0.86 = 2.58$ per execution. Consider a 4-place binary tree buffer and a 4-place linear buffer. The latter has a cost of $4 \times 1.0 = 4.0$, whereas the former has a cost of $2.58 + \frac{1}{2}(1.0 + 1.0) + 2.58 = 6.16$. The linear buffer is more energy efficient than its binary tree counterpart for small buffer sizes. As the size of the buffer increases, the binary tree buffer will eventually be more energy efficient than the linear buffer.

6 Conclusion

In this paper, we have shown how to estimate the energy dissipation of an asynchronous circuit based on the CSP specification of that circuit. This energy dissipation model can be used to compute a lower bound to the achievable energy dissipation for a given specification. This lower bound does not depend on the choice of implementation, rather it is derived from the expected input/output behavior of the circuit.

We have related the results derived from the CSP-based energy model to results derived from trace-theory. Trace theory was used to generalize the theorems presented in Section 3 to other design methodologies. Finally, we have shown how to use trace-theory and the statistics of the computation to derive energy-optimal CSP programs.

Acknowledgements

We would like to thank the anonymous referees for their helpful comments.

References

[1] Steven M. Burns and Alain J. Martin. Syntax-directed translation of concurrent programs into self-timed circuits. In J. Allen and F. Leighton, editors, *Proceedings of the Fifth MIT Conference on Advanced Research in VLSI*, pages 35–50. MIT Press, 1988.

[2] David L. Dill. Trace theory for automatic hierarchical verification of speed-independent circuits. In Jonathan Allen and F. Thomson Leighton, editors, *Advanced Research in VLSI: Proceedings of the Fifth MIT Conference*, pages 51–65. MIT Press, 1988.

[3] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[4] D. A. Huffman. A method for the construction of minimum redundancy codes. In *Proc. IRE*, volume 40, pages 1098–1101, 1952.

[5] Rajit Manohar. The entropy of traces in parallel computation. Submitted, IEEE Transactions on Information Theory.

[6] Alain J. Martin, Steven M. Burns, T. K. Lee, Drazen Borkovic, and Pieter J. Hazewindus. The first asynchronous microprocessor: the test results. *Computer Architecture News*, 17(4):95–110, June 1989.

[7] Farid Najm. Towards a high-level power estimation capability. In *Proceedings 1995 Symposium on Low Power Design*, pages 87–92. Association for Computing Machinery, April 1995.

[8] Lars Skovby Nielsen and Jens Sparsø. Low-power operation using self-timed and adaptive scaling of the supply voltage. In *1994 International Workshop on Low Power, Napa, California*, April 1994.

[9] C. Shannon. A mathematical theory of communication. In *Bell Systems Tech. J.*, volume 27, pages 379–423, 1948.

[10] José A. Tierno. *An Energy Complexity Model for VLSI Computations*. PhD thesis, California Institute of Technology, 1995.

[11] Kees van Berkel, Ronan Burgess, Joep Kessels, Ad Peeters, Marly Roncken, and Frits Schalij. A fully-asynchronous low-power error corrector for the DCC player. In *International Solid State Circuits Conference*, pages 88–89, February 1994.

[12] Jan L. A. van de Snepscheut. *Trace Theory and VLSI Design*, volume 200 of *Lecture Notes in Computer Science*. Springer-Verlag, 1985.