Low-Energy Asynchronous Memory Design

José A. Tierno   Alain J. Martin
California Institute of Technology
Pasadena, CA 91125

Abstract

We introduce the concept of energy per operation as a measure of performance of an asynchronous circuit. We show how to model energy consumption based on the high-level language specification. This model is independent of voltage and timing considerations. We apply this model to memory design. We show first how to dimension a memory array, and how to break up this memory array into smaller arrays to minimize the energy per access. We then show how to use cache memory and pre-fetch mechanisms to further reduce energy per access.

Keywords: Low-energy, low-power, asynchronous design, memory design.

1 Introduction

Present day portable computers run the most common interactive applications (word-processors, spreadsheets, windows, etc.) with no noticeable computation delay; weight and battery life have become more important than processing speed. These two factors are related by battery size: to operate the computer for a longer time without recharging, we need a larger, heavier, battery.

The limitation is therefore in the total amount of electric energy stored in that battery, that is available for operation. To extend the battery life, we have to make the computer more efficient in the way it uses this energy.

Electrical power dissipation has been used as a figure of merit for this type of application. It is convenient for synchronous circuits with no power management, where power dissipation is very much independent of the level of activity of the circuit. Asynchronous operation is better described in terms of reactive programs: energy is dissipated only when the circuit is active. As a consequence, asynchronous circuits can have remarkable energy performance [6, 7]. For asynchronous systems, a proper measure of performance is the "energy per operation." This metric measures the energy required to execute an instruction, fetch a piece of data from memory, service an interrupt, etc. To maximize the battery life, we can minimize the average energy per operation, that is, we maximize the number of instructions that we can execute with one battery charge.

Energy per operation is an additive quantity: given a computation described in terms of more elementary steps, we can calculate the energy required to execute that computation by adding the energy requirements of each step. In this way we can compare the energy efficiency of different algorithms that execute the same computation, independently of timing considerations. Comparison power consumptions would require some knowledge about timing (e.g. "so much power at so much throughput").

In this article we propose an energy model for asynchronous circuits based on the energy cost of data communications. This model is justified in terms of the physical implementation of a communication action, and the actual energy dissipation associated with that implementation.

As an example of the use of the energy model, we analyze the design of asynchronous memories. Memory subsystems are usually designed for speed and density, with secondary consideration given to energy. Memory is slow compared to processors, and high throughput is achieved through parallelism (wide data-words) and prediction (memory caching). These same design techniques can be used to improve energy performance; the design criteria are, however, different, and are explained in detail in this paper.

First we show how to partition a memory array to minimize access energy under the assumption that all addresses are equally probable. Second, we show how to use the statistics of long sequences of addresses to further reduce the average energy per access. These techniques result in a trade-off between area and energy per access. This analysis shows that conventional commercial architectures are not optimal from the point of view of energy efficiency.

2 Energy Index

The energy dissipation of a CMOS circuit is dependent on the supply voltage: the speed of operation and the energy required to charge capacitors increases at higher voltages. In order to evaluate the energy efficiency of a high-level circuit description, we need a measure of energy dissipation that is independent of the supply voltage. In this section, we derive such an index of performance, and use it in the next section to justify an energy model for asynchronous circuits based on the energy cost of communication actions.

2.1 Sources of energy dissipation

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. For $V_{GS} < V_{th}$, the channel conductance, g_c, can be modeled by [11]:

$$g_c = I_c \frac{q}{kT} \exp\left(\frac{q(V_{GS} - V_{th})}{kT}\right) \tag{2}$$

All these currents add up, and are responsible for an energy dissipation of the form

$$E_s = \int V_{DD}^2 I_c \frac{q}{kT} \exp\left(-\frac{qV_{th}}{kT}\right) dt \tag{3}$$

where $V_{GS} = 0$ is assumed. At the present state of the technology, energy dissipation due to leakage currents represents only a small fraction of the total power of a CMOS circuit.

Short-circuit currents originate in the short transients, as in the case of a CMOS inverter, when both pull-up and pull-down transistors conduct while the input signal switches between V_{thn} and $V_{DD} - V_{thp}$. This energy dissipation has the form [12]:

$$E_{sc} = \sum s_i \left(V_{DD} - 2V_{th}\right)^3 \tag{4}$$

where the s_i's are proportionality constants, and the sum is made over all transitions executed in one operation. Short-circuit currents also play a significant role in storing a value into a flip-flop built from cross-coupled inverters.

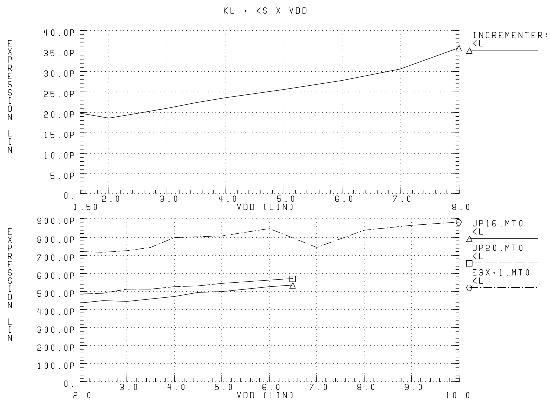

Figure 1: Graph of E_T/V_{DD}^2 against V_{DD} for a 4-bit counter (SPICE simulation), and for the 3x+1 engine, and the $1.6\mu m$ and $2.0\mu m$ processors.

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:

$$E_d = \sum_{C_i} n_i C_i \times V_{DD}^2 \tag{5}$$

where the C_i's are all the capacitors in the circuit, and n_i is the number of times the capacitor is switched in the execution of one operation. We rewrite Eq. 5 as:

$$E_d = K_L \times V_{DD}^2 \tag{6}$$

2.2 Linear Energy Model

Using Eqs. 4 and 6, and neglecting the effect of sub-threshold currents, we rewrite the energy equation as:

$$E_T = \left(K_L + K_S \frac{\left(V_{DD} - 2V_{th}\right)^3}{V_{DD}^2}\right) V_{DD}^2 \tag{7}$$

Outside the sub-threshold region, ($V_{DD} \gg V_{th}$), Eq. 7 simplifies to:

$$E_T = \left(K_L + K_S V_{DD}\right) V_{DD}^2 \tag{8}$$

Figure 1 shows E_T/V_{DD}^2 as a function of V_{DD} for a 4-bit counter (SPICE simulation), and for the Caltech Asynchronous Microprocessor and for a $3x+1$ engine (measurement). This figure shows that the linear approximation of Eq. 7 is indeed accurate.

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

As a first-order approximation we assume $K_s = 0$, and use K_L as the energy performance index.

3 Energy model for CHP programs

Our high-level description language is CHP (Communicating Hardware Processes) [5], which is similar to CSP [4]. The CHP specification of an asynchronous circuit corresponds very closely to its implementation; for each assignment, communication, function evaluation executed by the CHP program there will be a corresponding assignment, communication, function evaluation computed by the CMOS implementation. The CMOS implementation will dissipate energy only during the execution of the assignment, etc.. This energy can be assimilated to the energy required to execute the corresponding CHP statement. To calculate the energy required to execute a CHP program, we add the energy required to execute each statement in the trace of that program.

We would like to be able to map each statement into an energy performance index, independently of the other statements in the program. In general, it is not possible to do so; layout constraints make that the length—and therefore capacitance—of wires is affected by the connectivity of the whole circuit, not just the local connections. A detailed energy model would have to take into consideration the program as a whole, instead of individual statements.

The purpose of the model is, however, to study architectural trade-offs (e.g. compare bit-serial and parallel implementation of a function) or determine architectural parameters (e.g. determine the optimal width of a cache memory). A very detailed model, with a large number of parameters can be intractable, and not that much more accurate if those parameters are layout-dependent (and therefore not well known before the layout is finished). At the architectural design stage a simpler model is desirable.

The model proposed is based on the energy performance index. To each type of statement, we assign a capacitance that is representative of the energy that we would expect that operation to cost in a typical implementation. A full discussion of the possibilities and limitations of this model can be found in [10]

3.1 Communication

A CHP data communication involves two actions: first, copying the data into the wires that implement the communication channel, and second, copying the data from the communication channel into a register; the second part may not be present if the data is to be tested on the channel wires.

We assume that data communications are implemented with a four phase, dual-rail encoded protocol. The first action involves two transitions per bit; the second action involves one transition per bit in average. If the channel is one-to-one, the energy index of this communication will be proportional to the number of bits. If the channel is a bus (i.e. the channel has three or more ports), the capacitance of the wires will increase with the number of connections. To incorporate this effect, we scale the capacitance of the send action proportionally to the number of senders, and the capacitance of the receive action proportionally to the number of receivers.

3.2 Shared Variables

Variables shared by many processes are more expensive to implement than local variables. The value of those variables have to be known in many places, which increases the capacitance of all the related wires. To represent this cost, we scale the cost of writing into a variable proportionally to the number of processes that can read or write from that variable.

This cost has a number of important consequences. Even though the original specification of the circuit may not contain shared variables, some will appear after process decomposition. Also, guard evaluation may involve several tests on the same variable. The process decomposition presented above for choice statements will make that cost explicit by distributing the guard evaluation, one process per guard. The actual cost of an assignment is therefore not known until after process decomposition. We can, nevertheless, make an estimate of the worst case implementation of the assignment by scaling its cost proportionally to the number of times the variable is used in the program text.

3.3 Selection

The cost of selection is the difference in energy consumption between executing one statement from each of the following two programs:

$$PAR \equiv \langle \,\|\, i : 1..N : \ \texttt{*[[}\ G_i \longrightarrow A_i\ \texttt{]]} \rangle$$

and,

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

The second program can be transformed into the first program by adding state variables and an extra process:

$$\begin{array}{l} CHOOSE \equiv \\ \quad \langle\ \|\ i : 1..N : \\ \quad\ \ *[[\ \neg u \wedge G_i \longrightarrow u_i\uparrow; [u]; A_i; u_i\downarrow\ ]] \\ \quad \rangle\ \| \\ \quad\ \ *[[\langle\vee\ \ i : 1..N : u_i]; u\uparrow; \\ \quad\quad\ [\langle\wedge\ \ i : 1..N : \neg u_i]; u\downarrow \\ \quad\quad] \end{array}$$

Because this implementation of a selection is completely general, the cost of selection is at most the cost of an Or-gate. This cost scales proportionally to the log of the number of inputs.

3.4 Function evaluation

Function evaluation can hide part of the computation executed by the program; to incorporate that cost into the energy model, we have to make the evaluation of that function explicit in the CHP specification, or otherwise use a worst case cost for the evaluation of an arbitrary boolean function.

Given the program:

$$\ldots; F!f(x); \ldots$$

we want to express the cost of the evaluation of $f(x)$. To estimate the worst case cost we give a specific implementation for f and calculate the cost of that implementation based on the energy model described so far; that way we know that the cost of evaluating a function is consistent with the rest of the model.

If the range of x is $\{\mathrm{x}_1, \ldots, \mathrm{x}_n\}$, and $f(\mathrm{x}_i) = f_i$, we can express the function evaluation as:

$$\ldots; [\langle\ \ i : 1..n :\ \ x = \mathrm{x}_i \longrightarrow F!f_i\rangle]; \ldots$$

The cost of this program scales with n, which can be a large number. To obtain a more efficient implementation, we encode x as an array of $N = \lceil \log_2 n \rceil$ bits, and eliminate one bit from the function evaluation by currying:

$$\begin{array}{l} \ldots; \\ [\ \ \ x[0] \longrightarrow X_t!x[1..N-1];\ \ F!(F_t?) \\ [\!]\ \neg x[0] \longrightarrow X_f!x[1..N-1];\ \ F!(F_f?) \\ ]; \ldots \\ \\ \|\ *[X_t?y;\ \ F_t!f_t(y)] \\ \\ \|\ *[X_f?y;\ \ F_f!f_f(y)] \end{array}$$

From the previous decomposition we see that the cost of evaluating a function of N bits, $\mathrm{C}_f(N)$, is, at worst, the cost of communicating $N-1$ bits (X_{tf} channel), $K_c \times (N-1)$) plus the cost of evaluating an $N-1$ bit function, $\mathrm{C}_f(N-1)$ plus the cost of merging the F_t and F_f channels, K_M:

$$\mathrm{C}_f(N) = K_c \times (N-1) + \mathrm{C}_f(N-1) + K_M \qquad (9)$$

Solving for $\mathrm{C}_f(N)$,

$$\mathrm{C}_f(N) = K_c \frac{N(N-1)}{2} + \mathrm{C}_f(0) + NK_M \qquad (10)$$

This equation can be further refined. If the range of the function f has m different values, that can be expressed as an array of $M = \lceil \log_2 m \rceil$ bits, the cost of merging F_t and F_f can be expressed as $K_M = MK_m$, and we have:

$$\mathrm{C}_f(N) = K_c \frac{N(N-1)}{2} + \mathrm{C}_f(0) + NMK_m \qquad (11)$$

In general, the cost of evaluating a function of N inputs and M outputs, $\mathrm{K}_f(N, M)$ can be expressed as:

$$\mathrm{K}_f(N, M) \approx K_1 N^2 + K_2 NM \qquad (12)$$

4 Memory array

In CHP, a memory is an array, and reading from memory is one of the two operations: $x := M[a]$ or $X!M[a]$; writing to memory is one of the two operations $M[a] := y$ or $Y?M[a]$, where $M[a]$ is an array of n words of b bits. A program that describes a memory array with one read port and one write port is:

$$\begin{array}{l} MEM \equiv *[[\ \ \overline{R} \longrightarrow\ \ A?a;\ R!M[a] \\ \qquad\qquad\ [\!]\ \overline{W} \longrightarrow\ \ A?a;\ W?M[a] \\ \qquad\qquad]] \end{array}$$

The indexing $M[a]$ is removed by breaking up the memory array into a decoder and an array of registers:

$$\begin{array}{l} DECODER \equiv \langle\|\ \ i : 0..n-1 : \\ \qquad\qquad *[[\ \ \overline{R} \wedge (\overline{A?} = i) \longrightarrow\ \ A?; R_i! \\ \qquad\qquad\ \ [\!]\ \ \overline{W} \wedge (\overline{A?} = i) \longrightarrow\ \ A?; W_i! \\ \qquad\qquad]] \\ \qquad\ \rangle \\ ARRAY \equiv \langle\|\ \ i : 0..n-1 : \\ \qquad\qquad *[[\ \ \overline{R_i} \longrightarrow R_i \bullet R!x_i\ \ ]] \\ \qquad\quad \|\ \ *[[\ \ \overline{W_i} \longrightarrow W_i \bullet W?x_i\ \ ]] \\ \qquad \rangle \end{array}$$

Channel	Type	Width	Cost
A	1-to-n	$\log_2 n$	$K_A n \log_2 n$
R	n-to-1	b	$K_R nb$
W	1-to-n	b	$K_W nb$

Table 1: Cost of the communications involved in accessing an $n \times b$ memory array.

To read one word from the array, we have to execute an A communication (one sender, n receivers, $\log_2 n$ bits wide), an R_i communication (one sender, one receiver, data-less), and an R communication (n senders, one receiver, b bits wide). These costs are summarized in Table 1. The energy cost of reading one word is the sum of the energy costs of executing each of these communications, that is:

$$E_{1D,R}(n,b) = K_A n \log_2 n + K_{R_i} + K_R nb \qquad (13)$$

where K_A, K_{R_j}, and K_R are layout-dependent proportionality constants.

A one dimensional array is a viable solution only for small arrays; for large n, the energy cost scales like $n \log_2 n$. One way of improving on this cost is by mapping the one-dimensional array into a two-dimensional array (we verify this fact later). We represent the double indexing by splitting the address in two:

$$\begin{aligned} MEM \equiv *[[\ & \overline{R} \longrightarrow A?(a_w, a_l);\ R!M[a_l][a_w] \\ [\!]\ & \overline{W} \longrightarrow A?(a_w, a_l);\ W?M[a_l][a_w] \\ ]] & \end{aligned}$$

The first indexing is removed by extracting a row decoder:

$$\begin{aligned} DEC \equiv \langle \| \ & i : 0..l-1 : \\ & *[[\ \left(\overline{Al?} = i\right) \longrightarrow Al? \,\|\, S_i!\ ]] \\ \rangle & \end{aligned}$$

The second indexing is removed by extracting a column decoder:

$$\begin{aligned} & MUX \equiv \\ & \langle \| \ j : 0..w-1 : \\ & \quad *[[\ \overline{R} \wedge \left(\overline{Aw?} = j\right) \longrightarrow Aw? \,\|\, R!(R_j?) \\ & \quad [\!]\ \overline{W} \wedge \left(\overline{Aw?} = j\right) \longrightarrow Aw? \,\|\, W_j!(W?) \\ & \quad]] \\ & \rangle \end{aligned}$$

where l and w are such that $l \times w = n$. Finally, the register processes:

$$\begin{aligned} ARRAY \equiv \langle \| \ & i : 0..l-1 : j : 0..w-1 : \\ & *[[\ \overline{S_i} \wedge \overline{R_j} \longrightarrow S_i \bullet R_j!x_{ij}\ ]] \\ \| \ & *[[\ \overline{S_i} \wedge \overline{W_j} \longrightarrow S_i \bullet W_j?x_{ij}\ ]] \\ \rangle & \end{aligned}$$

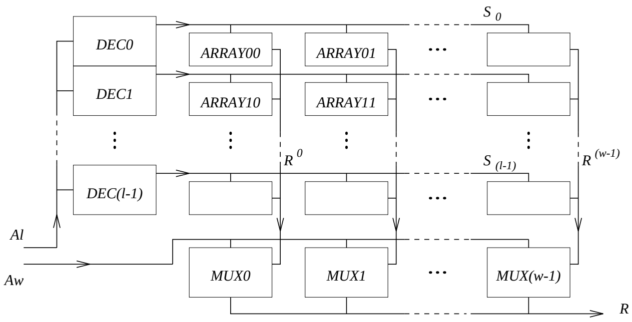

Figure 2: Process decomposition of *MEM* as a two-dimensional array. Only the channels corresponding to a read operation are shown.

Channel	Type	Width	Cost
Al	1-to-l	$\log_2 l$	$K_{Al} l \log_2 l$
Aw	1-to-w	$\log_2 w$	$K_{Aw} w \log_2 w$
S_i	1-to-w	dataless	$K_S w$
R_j	l-to-1	b	$K_{R_j} lb$
R	w-to-1	b	$K_R wb$
W_j	1-to-l	b	$K_{W_j} lb$
W	1-to-w	b	$K_W wb$

Table 2: Cost of the communications involved in accessing an $l \times w \times b$ memory array.

The process decomposition and channel interconnection are shown in Fig. 2.

In the following section we show how to choose l and w from a simple energy model.

4.1 Energy model and optimization

The energy cost of accessing one element of the array is calculated as the sum of the costs of the communications executed by the *DEC*, *MUX*, and *ARRAY* processes. A read from memory requires executing communication R (w senders, one receiver, b bits wide), communication Al (one sender, l receivers, $\log_2 l$ bits wide), communication Aw (one sender, w receivers, $\log_2 w$ bits wide), communication S_i (one sender, w receivers), and communication R_j (l senders, one receiver, b bits wide). Table 2 summarizes the energy costs for all communication actions in *MEM*.

The total energy cost E_r of reading a memory location is:

$$\begin{aligned} E_{2D,R}(n,b) \;=\; & K_{Al} l \log_2 l + K_{Aw} w \log_2 w + \\ & K_S w + K_{R_j} lb + K_R wb \qquad (14) \end{aligned}$$

We simplify Eq. 14 by assuming all constant equal to one. This approximation is acceptable for most technologies; if a more accurate model is needed, the parameters can be calculated from the layout, and the optimization is done with those values of the parameters.

$$E_{2D,R}(n,b) = l \log_2 l + w \log_2 w + (w+l)b + w \quad (15)$$

We minimize $E_{2D,R}$ with respect to l and w under the constraint $l \times w = n$, using Lagrange multipliers:

$$U = l \log_2 l + w \log_2 w + (w+l)b + w + \lambda(n - lw) \quad (16)$$

We take derivatives with respect to l, w, and λ:

$$\begin{aligned} \frac{\partial U}{\partial l} &= \log_2 l + \log_2 e + b - \lambda w && (17)\\ \frac{\partial U}{\partial w} &= \log_2 w + \log_2 e + b + 1 - \lambda l && (18)\\ \frac{\partial U}{\partial \lambda} &= n - lw && (19) \end{aligned}$$

Assuming that $\frac{1}{2}\log_2 n + b + \log_2 e \gg 1$, we solve for l, w, and λ, and $l_{opt} = w_{opt} = \sqrt{n}$. The optimum energy per access, $E_{2D,R}(n,b)$ is:

$$E_{2D,R}(n,b) = \sqrt{n}\,(\log_2 n + 2b + 1) \quad (20)$$

A memory designed for speed usually has $l = b \times w$ [1]. A completely square bit-array optimizes the access time per bit, but does not take into account the energy savings derived from selecting only the bits that are part of the desired word. This extra selection step takes time and area, and saves energy.

If we compare the optimal energy for a two dimensional array with the energy used by a one-dimensional array (assuming that all constants are equal to one) we get:

$$\frac{E_{1D}}{E_{2D}} = \frac{n(\log_2 n + b + 1)}{\sqrt{n}\,(\log_2 n + 2b + 1)} \approx \sqrt{n} \quad (21)$$

The number of words in a memory chip is usually very large, in the order of 2^{20}, making the two dimensional arrangement far better in energy. We can, in principle, generalize this argument to multidimensional arrays, to get an even greater improvement in energy per access. This cannot be done, however, by simply increasing the number of indices in the array. The memory has to be laid out on a 2-dimensional surface; mapping a multidimensional array on this surface will make all wires much longer, and the results will not be comparable with Eq. 20. In the next section we make that mapping explicit in the CHP program for the memory, so that a realistic energy model can be derived from that program.

4.2 Multi-bank memory array

We can further reduce the energy per access by breaking up the memory into several sub-arrays, so that only one of the smaller sub-arrays is accessed in each memory reference; this technique is also known as the divided word-line method [13]. We obtain the CHP for the multi-banked memory by applying a *divide-and-conquer* strategy to the *MEM* program.

```
MEM2B ≡ (MERGE || MEMO || MEME)

MERGE ≡
    *[[ R̄ ⟶ A?a;
              [ odd(a) ⟶ Ao!a/2 || R!(Ro?)
              ▯ even(a) ⟶ Ae!a/2 || R!(Re?)
              ]
      ▯ W̄ ⟶ A?a;
              [ odd(a) ⟶ Ao!a/2 || Wo!(W?)
              ▯ even(a) ⟶ Ae!a/2 || We!(W?)
              ]
    ]]
MEMO ≡
    *[[ R̄o ⟶ Ao?a; Ro!MO[a]
      ▯ W̄o ⟶ Ao?a; Wo?MO[a]
    ]]
MEME ≡
    *[[ R̄e ⟶ Ae?a; Re!ME[a]
      ▯ W̄e ⟶ Ae?a; We?ME[a]
    ]]
```

where ME and MO are $n/2 \times b$ arrays.

To read one word from $MEM2B$, we have to execute communication A (one sender, one receiver, $\log_2 n$ bits wide), communication R (two senders, one receiver, b bits wide), plus we have to execute either $MEMO$ or $MEME$. The energy cost of reading one word from a memory of size $n \times b$ can therefore be expressed as:

$$E_{2B,R}(n,b) = 2K_R b + K_A \log_2 n + E_{2B,R}(n/2, b) \quad (22)$$

Or, for $n = 2^N$,

$$E_{2B,R}(2^N, b) = E_{2B,R}(2^{N-1}, b) + 2K_R b + K_A N \quad (23)$$

We apply the same transformation to the sub-arrays, until the indexing is completely removed. We get:

$$E_{2B,R}(2^N, b) = E_{2B,R}(1,b) + 2K_R N b + K_A \frac{N(N+1)}{2} \quad (24)$$

Or, in terms of n,

$$E_{2B,R}(n,b) \approx 2K_R b \log_2 n + K_A \frac{\log_2 n(\log_2 n - 1)}{2} \quad (25)$$

Depending on the cost of merging the results from the two sub-arrays, it may be convenient to stop the divide-and-conquer process after fewer than N steps, and implement the remaining array as a two-dimensional array. After $N-J$ divide steps, the energy cost is:

$$E_{2B,R}(2^N, b) = E_{2D,R}(2^J, b) + 2K_R(N-J)b + K_A\frac{N(N+1) - J(J+1)}{2} \quad (26)$$

Minimizing Eq. 26 with respect to J we obtain the optimum bank size, $2^{J_{opt}}$.

For example, if all constants are equal to 1, $N = 2o$, and $b = 32$, we obtain an optimum for $J = 3$, $E_{2B,R}(2^{20}, 32) = 1493$. For $J = 20$ (no break-up in banks), the energy cost is $E_{2D,R}(2^{20}, 32) = 87040$.

Choosing the optimum number of banks requires more accurate knowledge of the constants in the energy equation. Some feed-back from the layout can help in determining those constants with sufficient accuracy.

5 Address prediction

The techniques described above try to minimize the energy cost of a single access to memory, under the assumption that all addresses are equally probable. In most applications, long address sequences are far from random: the past history of the address sequence is used to increase memory throughput [8, 9, 3]. The same type of information can be used to decrease the average energy per access.

The assumptions we make about the address sequence are *spacial* and *temporal locality* [2]. Spacial locality indicates that, once an address has been accessed, there is a strong probability that nearby addresses will be accessed in the near future. Temporal locality indicates that, once an address has been accessed, there is a strong probability that the same address will be accessed again in the near future.

Spacial locality is used by pre-fetch mechanisms. The cost per word of fetching a multi-word line from memory decreases with the number of words on the line (the energy cost of decoding the address is shared among more words). If we can predict that a sufficient number of words on a line will be used, we will be able to decrease the average cost of accessing a word in memory.

Temporal and spacial locality can be used to store a copy of the contents of the memory locations most likely to be needed in the future, in a small, fast, energy efficient memory. If the locality is strong enough, most of the memory references will be serviced by the small memory, with a corresponding improvement in energy performance.

5.1 Sequential access memory: spacial locality

Instruction memory accesses are, most of the time, accesses to consecutive memory locations. This fact can be exploited in two ways to make a more efficient memory in terms of speed and energy cost. First, the address does not need to be communicated all of the time, it can be calculated locally. Second, several consecutive words can be read in parallel at one time, thus reducing the number of memory references.

The following program describes a read-only memory with sequential access:

$MEMS \equiv$

$\quad *[[\ \overline{A} \longrightarrow A?a \ \ [\!]\ \overline{D} \longrightarrow D!M[a{+}{+}]\ ]]$

where M is an $n \times b$ array ($a{+}{+}$ means post-increment a with wrap-around). After an address a is sent to the memory, several data requests are executed. Sequencing between the A and D communications is maintained by the environment.

We can reduce the number of accesses to the array M by reading several words in parallel. We replace M by an $(n/m) \times mb$ array, MP. After receiving an address, the variable *line* is read from the array, and subsequent data requests are satisfied with data from *line*, until $a.w$ overflows, and new data is read into *line*.

$MEMR \equiv \ (PREF \ \| \ MEMP)$

$PREF \equiv$

$\quad *[[\ \overline{A} \longrightarrow A.w?a.w,\ B!A.l?,\ v\!\downarrow$

$\quad\quad [\!]\ \overline{D} \wedge v \longrightarrow D!line[a.w{+}{+}];$

$\quad\quad\quad\quad [a.w = 0 \longrightarrow v\!\downarrow$

$\quad\quad\quad\quad [\!]\ a.w \neq 0 \longrightarrow v\!\uparrow$

$\quad\quad\quad\quad]$

$\quad\quad [\!]\ \overline{D} \wedge \neg v \longrightarrow L?line,\ v\!\uparrow$

$\quad]]$

$MEMP \equiv$

$\quad *[[\ \overline{B} \longrightarrow B?b \ \ [\!]\ \overline{L} \longrightarrow L!MP[b{+}{+}]\ ]]$

Notice that *MEMP* has the same form as *MEMS*.

We compute next the average energy cost of reading a word from *MEMR*, and compare that cost with the energy cost of reading a word from a memory array. Let k be the number of consecutive memory references. To satisfy those k requests, the program *MEMR* has to execute: one $A.w$, $A.l$, and B-communication; k D-communications; about $\lceil \frac{k}{m} \rceil$ L-communications; about $\lceil \frac{k}{m} \rceil$ reads from an $(n/m) \times mb$

array (MP); and k reads from a $m \times b$ array (*line*). The energy cost of executing those k requests is:

$$\begin{aligned} k \times E_S &= K_A \log_2 n + K_B \log_2 \frac{n}{m} + \\ & k(E_R(m,b) + K_D) + \\ & \left\lceil \frac{k}{m} \right\rceil (E_R(n/m, mb) + K_L) \end{aligned} \quad (27)$$

where $E_R(n,b)$ is the energy cost of reading an $n \times b$ array. We take expected value on both sides of the equation, to obtain the expected energy cost of reading one word from *MEMR*, $E_S(n,m,b)$:

$$\begin{aligned} & E_S(n,m,b) = \\ & \quad \Bigg(K_A \log_2 n + K_B \log_2 \frac{n}{m} + \overline{k}(E_R(m,b) + K_D) + \\ & \quad \overline{\left\lceil \frac{k}{m} \right\rceil} (E_R(n/m, mb) + K_L) \Bigg) \Big/ \overline{k} \end{aligned} \quad (28)$$

To optimize the previous equation, we use Eq. 26 for $E_R(n/m, mb)$; $\overline{k}$ and $\overline{\lceil \frac{k}{m} \rceil}$ are determined from program traces.

For example, if all constants are equal to 1, $n = 2^{20}$, $b = 32$, $\overline{k} = 8$, we obtain the minimum energy cost for $m = 8$, $E_S = 1134$. Compared to the minimum energy cost of accessing a memory array, $E_R = 1493$, we do not obtain a significant improvement. However, the optimal block size for this parameter set is 8 words per block; with this block size, most of the silicon area occupied by the memory will be dedicated to routing of data and address, resulting in very poor memory density. We can choose a sub-optimal block size to improve in density; for a block size of 2^{10} words, we get $E_R = 3195$, and $E_S = 2590$. The pre-fetch mechanism allows us to use a denser memory with a smaller energy penalty.

5.2 Memory with cache: temporal locality

If the energy per access of a memory of size n is $E_M(n)$, and h is the hit ratio of the cache (that is, the fraction of addresses that are found in the cache), then the average energy per access of a system consisting of a memory of size n and a cache of size c is:

$$E_s = E_M(c) + (1-h)E_M(n) \quad (29)$$

and the reduction in energy per access, ρ, is

$$\rho = \frac{E_s}{E_M(n)} = 1 - h + \frac{E_M(c)}{E_M(n)} \quad (30)$$

We can replace E_M with one of the energy models previously derived. We use as a model for h, $h = \left(\frac{c}{n}\right)^p$, $0 < p \leq 0.5$, The shape of this curve corresponds approximately to the dependency of the hit ratio with cache size [8]; p is a parameter to be determined from measurements of cache statistics.

According to Eqs. 20 and 25, $\log_2^2 n \leq E_M(n) \leq \sqrt{n}\log_2 n$. We calculate next the possible energy savings derived from the use of a cache in the case $E_M(n) = \sqrt{n}$.

$$\rho = 1 - \left(\frac{c}{n}\right)^p + \sqrt{\frac{c}{n}} \quad (31)$$

Using $r = \frac{c}{n}$, and taking derivatives with respect to r, we find ρ_{opt}:

$$\rho_{opt} = 1 - (2p)^{\frac{2p}{1-2p}} + (2p)^{\frac{1}{1-2p}} \quad (32)$$

Usual values of p are in the range $0.01 \leq p \leq 0.1$ [8]. In this range, we get $0.1 \leq \rho_{opt} \leq 0.5$.

From the previous results we conclude that a cache designed for low-energy has to optimize the hit ratio at relatively small cache sizes, to make p as small as possible. In general, caches with good hit ratios use very complicated architectures, which make the energy cost of a cache access high. Fully associative caches, for example, require that a comparison be made for each line in the cache for every access, thus cancelling the energy advantage of a higher hit ratio. The hit ratio can be increased at the expense of added delay, or by specializing the cache to specific address sequence types (instruction memory references, vectors, I/O, etc.).

6 Conclusions

In this paper we have shown how to estimate the energy-per-operation cost of a CMOS asynchronous circuit from its CHP specification. This technique allows us to exploit early in the design process the trade-off between energy, area, and delay. Ultimately, it allows us to get to a circuit architecture more suited to low-energy design.

The energy model derived from the high-level specification of a circuit is, by its very nature, only approximate. It represents the energy complexity of the algorithm used to solve the problem at hand, under the assumption that there is a strong correlation between this energy complexity and the actual energy dissipation of the circuit.

We have presented several memory designs, and we have shown how to choose the design parameters to obtain the optimum energy cost. These results show that commercial memory designs, optimized for delay and density, can be greatly improved in energy performance.

Cache memory and pre-fetch mechanisms also improve the energy cost. Pre-fetch can be particularly efficient for instruction-memory references; the effectiveness of a cache is a little more limited.

Acknowledgments

The research described in this paper was sponsored by the Advanced Research Projects Agency, ARPA Order number 6202, and monitored by the Office of Naval Research under contract number N00014-87-K-0745.

References

[1] S. T. Chu, J. Dikken, C. D. Hartgring, F. J. List, J. G. Raemaekers, S. A. Bell, B. Walsh, and R. H. W. Salters. A 25-ns low-power full-CMOS 1-Mbit (128K×8) SRAM. *IEEE J. of Solid-State Circuits*, SC-23(5):1078–1084, October 1988.

[2] P. J. Denning. On modeling program behavior. *Proc. Spring Joint Computer Conference*, 40:937–944, 1972.

[3] J. L. Hennessy and D. A. Patterson. *Computer Architecture A Quantitative Approach*, chapter 8, pages 404–425. Morgan Kaufmann Publishers Inc., 1990.

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

[5] Alain J. Martin. Compiling communicating processes into delay-insensitive VLSI circuits. *Distributed Computing*, 1(4):226–234, 1986.

[6] Alain J. Martin, Steven M. Burns, T. K. Lee, Drazen Borkovic, and Pieter J. Hazewindus. The design of an asynchronous microprocessor. In Charles L. Seitz, editor, *Advanced Research in VLSI: Proceedings of the Decennial Caltech Conference on VLSI*, pages 351–373, 1989.

[7] 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.

[8] A. J. Smith. Cache memories. *Computing Surveys*, 14(3):473–530, September 1982.

[9] H. S. Stone. *High-Performance Computer Architecture*, chapter 2, pages 21–69. Addison Wesley, 1989.

[10] J. A. Tierno. *Architectural Strategies for Low-Energy Asynchronous Design.* PhD thesis, California Institute of Technology, 1994.

[11] R. R. Troutman. Subthreshold design considerations for insulated gate field effect transistors. *IEEE J. Solid State Circuits*, SC-9:55–60, April 1974.

[12] H. J. M. Veendrick. Short circuit dissipation of static CMOS circuitry and its impact on the design of buffer circuits. *IEEE Journal of Solid-State Circuits*, SC-19(4):468–473, August 1984.

[13] M. Yoshimoto, K. Anami, H. Shinohara, T. Yoshihara, T. Takagi, and T. Nakano. A divided word-line structure in the static RAM and its application to a 64K full CMOS RAM. *IEEE J. Solid-State Circuits*, SC-18(5):479–485, October 1983.

Appendix: notation

The notation is based on Hoare's CSP [4]. A full description of the notation and its semantics can be found in [5]. We give here a short and informal description.

Assignment: $a := b$ stands for "assign the value of b to a."

Boolean assignments:

$b\uparrow$ stands for $b :=$ **true**,

$b\downarrow$ stands for $b :=$ **false**.

Selection:

$$[G_1 \longrightarrow S_1 \,[\!]\, \ldots \,[\!]\, G_n \longrightarrow S_n]$$

where the G_i's are boolean expressions (guards) and the S_i's are program parts ($G_i \rightarrow S_i$ is a "guarded command"). The execution of this command correspond to waiting until one of the guards is **true**, and then executing one of the statements with a **true** guard. The notation $[G]$ is short-hand for $[G \rightarrow \textbf{skip}]$.

Repetition:

$$*[G_1 \longrightarrow S_1 \,[\!]\, \ldots \,[\!]\, G_n \longrightarrow S_n]$$

The execution of this command corresponds to choosing one of the true guards, executing the corresponding statement, and repeating until all guards are found to be false, in which case the command terminates. The notation $*[S]$ is short-hand for $*[\textbf{true} \rightarrow S]$.

Communication:

Send: $X!u$ means send the value of u over channel X.

Receive: $Y?v$ means receive a value over channel Y and store it in variable v.

Probe: The boolean expression $\overline{X}$ is **true** iff a communication over channel X can complete without suspending.

Composition operators:

Sequential composition: $S_1 ; S_2$.

Parallel composition: $S_1 \| S_2$.

Coincident composition of communication actions: $X \bullet Y$ (both communication actions complete at the same time).