Instruction Balance, Energy Consumption and Program Performance

Tao Li      Chen Ding
Computer Science Department
University of Rochester
Rochester, New York
{taoli,cding}@cs.rochester.edu

December 2000
Revised February 2001

Technical Report UR-CS-TR-739

Abstract

A computer processor consists of multiple components such as functional units, cache and main memory. At each moment of execution, a program may have a varied amount of work for each component. Recent research has exploited this imbalance to save energy by slowing the components that have a lower load. Symmetrical to reconfiguring hardware is reorganizing software. For each program segment, we can alter its demands for different components by reordering program instructions. In other words, we can change the balance of program loads in different parts of a program. This paper explores the theoretical lower bound of the energy consumption assuming that both a program and a machine are fully adjustable. It shows that a program with constant load balances always consumes less power than the same program with uneven balances. In addition, the paper examines the relation between energy consumption and program performance. It shows that as far as program reordering is concerned, reducing power is a more complex problem than improving performance.
1 Introduction

Many devices for personal and network computing are portable devices powered by batteries. Higher energy efficiency would allow for smaller batteries, lower device weight and longer operation. Therefore, managing the energy consumption of portable processors has become important because it directly leads to lower cost and better service. In addition, power reduction helps other computing platforms by reducing packaging complexity and cost. As computing devices permeate our daily life, saving energy has broad benefits to the society and environment.

Energy is consumed by all parts of a computer system, including functional units, cache and main memory. For each component, the power usage is largely determined by its operating speed. On most machines, the hardware configuration is fixed. When a program does not utilize all available capacity, some components are underutilized and consequently waste energy. The recent interest in energy efficiency has prompted rapid development of reconfigurable processors, which can adjust hardware speed to match the dynamic demand of applications. For example, when CPU frequently waits for slow memory loads, the machine can scale down the CPU frequency as well as voltage. This technique is called dynamic scaling. It has already found its way into a commercial processor, Transmeta Crusoe [6]. Given the improvement in hardware, a natural question is whether software can be adapted to better exploit the new reconfigurable processors.

For many years, programs have been analyzed and optimized based on the effect on performance. One effective optimization is to reduce program demand for hardware by eliminating redundant program instructions and memory access. While the fastest instructions are those that do not exist, those are also the most energy efficient. Although demand reduction saves power, its benefit does not depend on the support of reconfigurable hardware. The question still remains whether software can take advantage of the dynamic features in emerging energy-efficient machines such as Crusoe. Next, we demonstrate that such possibility exists. In particular, we show that demand reordering can make use of reconfigurable hardware and save more energy.

A Motivating Example

For illustration purpose, we assume a simple machine model with only two functional units for integer and floating-point operations. We also assume that each program is a sequence of blocks of instructions, instructions within each blocks are fully parallel but each block must be finished before the next block. Part (a) of Figure 1 shows a program with two blocks. We call the ratio of integer operations (fp op) to floating-point operations (int op) the instruction balance. The two blocks in Part (a) have different instruction balances. In Part (b), we reorder the program and make the instruction balances identical, assuming that the reordering is legal. Table 1 shows the energy consumption for both programs.

Table 1 shows three configurations: original program without and with hardware dynamic scaling, and the balanced program with dynamic scaling. Each configuration includes three rows: two for the two instruction blocks and one for their total. The data for each configuration are listed in columns. The third column lists the frequency used by integer and floating-point units, fint and ffp. The fourth column shows the execution time, t, which is the number of operations divided by the operating frequency. The total energy, E, shown in the last column, is summed for two units. Each unit consumes energy tfpc, where t is the time, f is the frequency, and c is a architectural dependent constant. The reason for this cubic function is explained in Section 2.

In the first configuration, all units run at the peak speed, fmax. With dynamic scaling in the second configuration, only one unit runs at the peak speed, the other unit runs at a lower frequency, ½fmax. For the balanced program in the third configuration, both units run at a lower speed, ½fmax. The three rows
labeled with “total” give the overall speed and energy consumption. All three configurations have the same execution time, $\frac{t_{\text{max}}}{2}$. The energy consumption, however, is significantly different. Dynamic scaling saves a little less than half of the energy (40%) compared to no scaling; program re-balancing further reduces the energy consumption by over a half, requiring only 24% of the initial energy.

This example has shown that program reordering can achieve significant power saving, even when we cannot reduce the number of instructions. In this example, reordering saved over 53% than that of the hardware scaling alone.

In the rest of this report, we present the theoretical lower bound of the energy consumption assuming that both a program and a machine are fully adjustable. Section 2 defines the program and machine model and formulates the concept of instruction balance. Section 3 proves the central theorem, which characterizes the relation between instruction balance and energy consumption. Section 4 extends the proof to a more general machine model. Section 5 analyzes the relation between energy consumption and program performance. Finally, Section 6 reviews related work and Section 7 concludes.

## 2 Program and Machine Model

This section describes a simplified system model. Since we intend to find the highest energy efficiency possible, we impose the least restrictions on our model. To simplify the presentation, we will also assume a machine with only two functional units running at any non-negative frequency. The extended theorems that
include $N$ units and discrete frequencies are given in Section 4.

**Program Model** A program, $P$, is a sequence of instruction blocks, $B_1, B_2, ..., B_n$. Each block, $B_i$, is a pair $(a_i, b_i)$, where $a_i$ is the number of integer operations and $b_i$ is the number of floating-point operations in the block. We assume two properties for the $i$th instruction block. First, different types of operations inside a block can be executed independently, so the integer and floating-point units can operate in an independent, uniform frequency. Second, a processor must finish all instructions of one block before starting on the next block. This program model is more general than it seems to be. Indeed, we can view each program by its execution trace and treat the set of instructions executed at each machine cycle as a block.

The *instruction balance* for a block $B_i$ is the ratio $\frac{a_i}{b_i}$. If the balance of all blocks is the same, that is, for any $i$ and $j$, $\frac{a_i}{b_i} = \frac{a_j}{b_j}$, we say the program has constant instruction balances. We call such a program a *balanced program*. The condition can be rewritten as $a_i = \frac{b_i}{b_j}$. The second format is more convenient when we extend the formulation to instructions of more than two types in Section 4. We note that constant balances are not necessarily unit balances. The number of integer and floating-point operations do not need to be the same. In fact, a balanced program can have any number of instructions in each type. Finally, to find the theoretical maximum, we assume that a compiler can freely move instructions from one block to another but cannot eliminate any instruction in any block.

**Machine Model** A machine consists of multiple functional units whose frequency can be independently adjusted by hardware to match software demand (i.e. dynamic voltage and frequency scaling). The total energy cost is the power consumption of each unit multiplied by the execution time. For each functional unit, we use the power model given by Burd and Brodersen [2]. Burd and Brodersen divided CMOS power into static and dynamic dissipation. Static power due to bias and leakage currents can be made insignificant by improved circuit design. Dynamic power, which dominates overall power, is proportional to $V^2 \cdot f \cdot C$ where $V$ is the supply voltage, $f$ is the clock speed and $C$ is the effective switching capacitance [2]. Here we assume that the voltage can be scaled linearly with frequency. Therefore, power consumption is a cubic function of frequency, that is, for unit $i$, $P_i \approx f^3 c_i$, where $c_i$ is an architectural dependent constant. We further assume that all $c_i$s are identical.

The above machine and program model is too simple to model a real system. Most program instructions cannot be arbitrarily reordered, and no machine consists of identical functional units. Furthermore, scaling between voltage and frequency is not linear [12]. We use the simplified model in the following theoretical study because it allows for the most freedom in software reorganization and gives a closed formula for energy consumption.

# 3 Instruction Balance and Energy Consumption

We now present the basic theorem: given any program, the order of its instructions is optimal for energy when it has the same instruction balance in all blocks. To prove, we show that given any program with uneven instruction balances, its balanced counterpart can finish the execution at the same time while consuming less energy. We next formulate the energy consumption of an arbitrary program and its balanced counterpart.

## 3.1 Problem Formulation

Given a program, $P = (B_1, ..., B_i, ..., B_n), B_i = (a_i, b_i), i = 1, ..., n$. Let $f$ be the maximum frequency. The original energy consumption $E_{original}$ with dynamic scaling can be computed in the following three steps.

- The execution time of each block, $t_i = \frac{M_i}{f}$ where $M_i = \max(a_i, b_i)$, $a_i$ and $b_i$ are the numbers of
instructions, $f$ is the maximal frequency of functional units.

- The power consumption of each block, $P_i = f^3 + \left(\frac{m_i}{M_i} f\right)^3$, where $m_i = \min(a_i, b_i)$.
- The program energy consumption,

\[
E_{\text{original}} = \sum_{i=1}^{n} t_i P_i = \sum_{i=1}^{n} \frac{M_i}{f} \left(f^3 + \left(\frac{m_i}{M_i} f\right)^3\right) = \sum_{i=1}^{n} \frac{M_i^3 + m_i^3}{M_i^2} f^2 = \sum_{i=1}^{n} \frac{a_i^3 + b_i^3}{M_i^2} f^2
\]

If we re-balance the program, i.e. transforming the program into $P' = (B_1', ..., B_n')$, $B_i' = (a_i', b_i')$, where $\frac{a_j'}{b_j'} = \frac{A}{B}$, $A = \sum_{i=1}^{n} a_i$, and $B = \sum_{i=1}^{n} b_i$. Suppose $P'$ runs in the same time as $P$, which is $t_{\text{total}} = \sum_{i=1}^{n} t_i = \sum_{i=1}^{n} \frac{M_i}{f}$. The following three steps compute the energy consumption $E_{\text{balanced}}$ for the transformed program, $P'$.

- The frequency of the integer unit, $f_{\text{int}} = \frac{A}{t_{\text{total}}} = \frac{A}{\sum_{i=1}^{n} M_i}$
- The frequency of the floating-point unit: $f_{\text{float}} = \frac{B}{t_{\text{total}}} = \frac{B}{\sum_{i=1}^{n} M_i}$
- The energy consumption:

\[
E_{\text{balanced}} = t_{\text{total}} (f_{\text{int}}^3 + f_{\text{float}}^3) = \frac{A^3 + B^3}{(\sum_{i=1}^{n} M_i)^2} = \frac{(\sum_{i=1}^{n} a_i)^3 + (\sum_{i=1}^{n} b_i)^3}{(\sum_{i=1}^{n} M_i)^2} f^2
\]

We now remove the common positive term $f^2$ from $E_{\text{original}}$ and $E_{\text{balanced}}$. The following theorem states their inequality and the condition for equality.

### 3.2 The Theorem and its Proof

**Theorem 1** The following inequality holds.

\[
\sum_{i=1}^{n} \frac{a_i^3 + b_i^3}{M_i^2} \geq \frac{(\sum_{i=1}^{n} a_i)^3 + (\sum_{i=1}^{n} b_i)^3}{(\sum_{i=1}^{n} M_i)^2}
\]

where $a_i, b_i \geq 0$, $\sum_{i=1}^{n} a_i > 0$, $\sum_{i=1}^{n} b_i > 0$, and $M_i \geq \max(a_i, b_i) > 0$. The equality holds when and only when $\frac{a_i}{b_i} = \frac{a_j}{b_j}$ for all $i$ and $j$.

The theorem says that the energy consumption of any program, represented by the left-hand side formula, is always greater than or equal to the energy consumption of its balanced self, represented by the right-hand side formula, assuming they have the same execution time. Therefore, balancing instructions can always save energy for any unbalanced program. The theorem is actually a bit stronger than necessary. It assumes $M_i \geq \max(a_i, b_i)$, while our formulation needs $M_i = \max(a_i, b_i)$. This generalization makes the proof easier to present, and it is necessary for proving the extended theorem for $N$ units in Section 4.

We prove the theorem by induction. We first reduce the case of $n + 1$ to the case of $n = 2$. We then use calculus method to prove the inequality for $n = 2$.

If $n = 1$, then the inequality holds trivially. Now suppose the inequality holds for $n$.

\[
\sum_{i=1}^{n} \frac{a_i^3 + b_i^3}{M_i^2} \geq \frac{(\sum_{i=1}^{n} a_i)^3 + (\sum_{i=1}^{n} b_i)^3}{(\sum_{i=1}^{n} M_i)^2}
\]

5
We need to prove
\[
\sum_{i=1}^{n+1} \frac{a_i^3 + b_i^3}{M_i^2} \geq \frac{(\sum_{i=1}^{n+1} a_i^3) + (\sum_{i=1}^{n+1} b_i^3)}{(\sum_{i=1}^{n+1} M_i)^2}
\]

Since
\[
\sum_{i=1}^{n+1} \frac{a_i^3 + b_i^3}{M_i^2} = \sum_{i=1}^n \frac{a_i^3 + b_i^3}{M_i^2} + \frac{a_{n+1}^3 + b_{n+1}^3}{M_{n+1}^2}
\]
Hence by induction hypothesis,
\[
\sum_{i=1}^{n+1} \frac{a_i^3 + b_i^3}{M_i^2} = \sum_{i=1}^n \frac{a_i^3 + b_i^3}{M_i^2} + \frac{a_{n+1}^3 + b_{n+1}^3}{M_{n+1}^2} \geq \frac{(\sum_{i=1}^n a_i^3) + (\sum_{i=1}^n b_i^3)}{(\sum_{i=1}^n M_i)^2} + \frac{a_{n+1}^3 + b_{n+1}^3}{M_{n+1}^2}
\]
So we need to show that
\[
\frac{(\sum_{i=1}^n a_i^3) + (\sum_{i=1}^n b_i^3)}{(\sum_{i=1}^n M_i)^2} + \frac{a_{n+1}^3 + b_{n+1}^3}{M_{n+1}^2} \geq \frac{(\sum_{i=1}^n a_i + a_{n+1}^3) + (\sum_{i=1}^n b_i + b_{n+1}^3)}{(\sum_{i=1}^n M_i + M_{n+1})^2}
\]
Now let \( a' = \sum_{i=1}^n a_i, b' = \sum_{i=1}^n b_i, M' = \sum_{i=1}^n M_i \), clearly we have \( M' \geq \text{max}(a', b') \) since \( M_i \geq \text{max}(a_i, b_i), i = 1, \ldots, n \). So we need to show that
\[
\frac{a'^3 + b'^3}{M'^2} + \frac{a_{n+1}^3 + b_{n+1}^3}{M_{n+1}^2} \geq \frac{(a' + a_{n+1})^3 + (b' + b_{n+1})^3}{(M' + M_{n+1})^2}
\]
Hence it is sufficient to prove that
\[
\frac{a_i^3 + b_i^3}{M_i^2} + \frac{a_{n+1}^3 + b_{n+1}^3}{M_{n+1}^2} \geq \frac{(a_i + a_{n+1})^3 + (b_i + b_{n+1})^3}{(M_i + M_{n+1})^2}
\]
where \( M_i \geq \text{max}(a_i, b_i), M_2 \geq \text{max}(a_2, b_2) \). Now the case of \( n = n + 1 \) is equivalent to the case of \( n = 2 \).

Next, we split the inequality into two parts.
\[
\frac{a_i^3}{M_i^2} + \frac{a_2^3}{M_2^2} \geq \frac{(a_i + a_2)^3}{(M_i + M_2)^2}
\]
\[
\frac{b_i^3}{M_i^2} + \frac{b_2^3}{M_2^2} \geq \frac{(b_i + b_2)^3}{(M_i + M_2)^2}
\]
Since the above two inequalities are equivalent, so we only need to prove one of them. Now we prove the first one in the following lemma. Note that the requirement of \( M_1 \) and \( M_2 \) is relaxed: they can be any positive number and do not have to be greater than \( a_1 \) and \( a_2 \).

**Lemma 3.1** \( \frac{a^3}{M_1^2} + \frac{a^3}{M_2^2} \geq \frac{(a_1 + a_2)^3}{(M_1 + M_2)^2} \), where \( a_1, a_2 \geq 0 \) and \( M_1, M_2 > 0 \).

**Proof:** We first convert \( M_2 \) to be 1 with the following transformation:
\[
\frac{1}{M_2^2} \left( \frac{a_i^3}{M_i^2} + \frac{a_2^3}{M_2^2} \right) \geq \frac{(a_i + a_2)^3}{(M_i + 1)^2} \times \frac{1}{M_2^2}
\]
let \( M_1 = \frac{M_1}{M_2} \), so we have

6
\[
\frac{a_1^3}{M_1^2} + \frac{a_2^3}{1^2} \geq \frac{(a_1 + a_2)^3}{(M_1 + 1)^2}
\]

So now let \( M = M_1 \), we prove that
\[
\frac{a_1^3}{M^2} + \frac{a_2^3}{1^2} \geq \frac{(a_1 + a_2)^3}{(M + 1)^2}
\]
\[\iff (M + 1)^2(a_1^3 + M_1 a_2^3) \geq M_1^2(a_1 + a_2)^3
\]
\[\iff (2M^{-1} + M^{-2})a_1^3 + (2M + M^2)a_2^3 \geq 3a_1 a_2(a_1 + a_2)
\]

The last inequality holds as shown in the following lemma.

**Lemma 3.2** \((2M^{-1} + M^{-2})a_1^3 + (2M + M^2)a_2^3 \geq 3a_1 a_2(a_1 + a_2)\), where \(a_1, a_2, b_1, b_2 \geq 0\), and \( M > 0 \).

**Proof:** If \(a_1 = 0\) or \(a_2 = 0\), the inequality holds trivially. Now we assume that \(a_1, a_2 > 0\). Define a function \( f(M) = (2M^{-1} + M^{-2})a_1^3 + (2M + M^2)a_2^3 \), where \(M, a_1, a_2 > 0\). We will show that the function \(f(M)\) achieve its minimum value at \(\frac{a_1}{a_2}\), when \(M > 0\). Note that \( f\left(\frac{a_1}{a_2}\right) = 3a_1 a_2(a_1 + a_2)\). In other words,
\[
f(M) \geq f\left(\frac{a_1}{a_2}\right) = 3a_1 a_2(a_1 + a_2)
\]

The first derivative is \(f'(M) = (-2M^{-2} - 2M^{-3})a_1^3 + (2 + 2M)a_2^3\).

The second derivative is \(f''(M) = (4M^{-5} + 6M^{-4})a_1^3 + 2a_2^3 > 0\).

Since \(f''(M) > 0\), so \(f'(M)\) is increasing. Let \(f'(M) = 0\), we get \(M = -1\) and \(M = \frac{a_1}{a_2}\). Note that \(M > 0\), so we only have \(f\left(\frac{a_1}{a_2}\right) = 0\). Since \(f'(M)\) is increasing, hence \(f'(M) \leq 0\) on \((0, \frac{a_1}{a_2}]\) and \(f'(M) \geq 0\) on \([\frac{a_1}{a_2}, \infty)\).

Therefore, \(f(M) \geq f\left(\frac{a_1}{a_2}\right), \forall M > 0\) and Lemma 3.1 holds. The theorem then follows from Lemma 3.1.

4 Extensions to Machine Model

The basic theorem assumes a machine with two functional units that operate on a frequency of any non-negative rational number. This section removes these two restrictions and extends the theorem to machines with more than two functional units that operate on a set of pre-determined frequencies.

4.1 Multiple Functional Units

First we generalize the definition of the instruction balance and then the theorem.

**Definition 4.1** Given a program \(P\):

\[
P = (B_1, ..., B_i, ..., B_n), B_i = (a_{i1}, ..., a_{im}), i = 1, ..., n
\]

The instruction balance for each block \(B_i\) is a \(m\)-tuple \((a_{i1}, ..., a_{im})\). A program is said to have constant instruction balances if \(\frac{a_{i1}}{a_{j1}} = \cdots = \frac{a_{im}}{a_{jm}}\), for any \(B_i, B_j\).

**Theorem 2** (Generalization of Theorem 1 for \(N\) functional units):

\[
\sum_{i=1}^{n} \frac{a_{i1}^3 + \cdots + a_{im}^3}{M_i^2} \geq \frac{\left(\sum_{i=1}^{n} a_{i1}\right)^3 + \cdots + \left(\sum_{i=1}^{n} a_{im}\right)^3}{\left(\sum_{i=1}^{n} M_i\right)^2}
\]

where \(a_{i1}, ..., a_{im} \geq 0\), and \(M_i \geq \max(a_{i1}, ..., a_{im}) > 0\).
We can prove the generalized theorem by induction on \(m\). The case of \(m = 2\) holds from theorem 1. Now suppose
\[
\sum_{i=1}^{n} \frac{a_{i1}^3 + a_{i2}^3 + \cdots + a_{im}^3}{M_i^2} \geq \frac{(\sum_{i=1}^{n} a_{i1})^3 + (\sum_{i=1}^{n} a_{i2})^3 + \cdots + (\sum_{i=1}^{n} a_{im})^3}{(\sum_{i=1}^{n} M_i)^2}
\]

We want to show
\[
\sum_{i=1}^{n} \frac{a_{i1}^3 + a_{i2}^3 + \cdots + a_{im+1}^3}{M_i^2} \geq \frac{(\sum_{i=1}^{n} a_{i1})^3 + (\sum_{i=1}^{n} a_{i2})^3 + \cdots + (\sum_{i=1}^{n} a_{im+1})^3}{(\sum_{i=1}^{n} M_i)^2}
\]

Since \(M_i \geq \max\{a_{i1}, ..., a_{im}, a_{im+1}\} \geq \max\{a_{im}, a_{im+1}\}\), so by induction hypothesis we have
\[
\sum_{i=1}^{n} \frac{a_{i1}^3 + a_{i2}^3 + \cdots + a_{im}^3 + a_{im+1}^3}{M_i^2} \geq \sum_{i=1}^{n} \frac{a_{i1}^3 + a_{i2}^3 + \cdots + a_{im}^3}{M_i^2} + \sum_{i=1}^{n} \frac{a_{im+1}^3}{M_i^2}
\]

Note that \(\sum_{i=1}^{n} \frac{a_{im+1}^3}{M_i^2} = \sum_{i=1}^{n} \frac{a_{im+1}^3 + b_{im+1}^2}{M_i^2} = \sum_{i=1}^{n} \frac{a_{im+1}^3 + b_{im+1}^2}{M_i^2} \geq \sum_{i=1}^{n} \frac{a_{im+1}^3}{M_i^2} = \sum_{i=1}^{n} \frac{a_{im+1}^3}{M_i^2} \geq \sum_{i=1}^{n} \frac{a_{im+1}^3}{M_i^2}
\]

Now we can apply Theorem 1 again and get
\[
\sum_{i=1}^{n} \frac{a_{im+1}^3 + b_{im+1}^2}{M_i^2} \geq \frac{(\sum_{i=1}^{n} a_{im+1})^3 + (\sum_{i=1}^{n} b_{im+1})^3}{(\sum_{i=1}^{n} M_i)^2} = \frac{(\sum_{i=1}^{n} a_{im+1})^3}{(\sum_{i=1}^{n} M_i)^2}
\]

Hence we have
\[
\sum_{i=1}^{n} \frac{a_{i1}^3 + a_{i2}^3 + \cdots + a_{im}^3 + a_{im+1}^3}{M_i^2} \geq \frac{(\sum_{i=1}^{n} a_{i1})^3 + (\sum_{i=1}^{n} a_{i2})^3 + \cdots + (\sum_{i=1}^{n} a_{im})^3 + (\sum_{i=1}^{n} a_{im+1})^3}{(\sum_{i=1}^{n} M_i)^2}
\]

So the generalized theorem holds.

The generalized theorem says that if we have multiple functional units that can run on different frequencies at the same time, then a balanced program would consume less energy than its counterpart with uneven instruction balances, assuming both have the same execution time.

### 4.2 Discrete Operating Frequencies

So far we have assumed that a processor can operate on any clock rate. However, the number of frequency choices or valid frequencies is limited on a real machine. The optimal frequency, as determined by the instruction balance and execution time, may lie between two valid frequencies. The solution in this case is to alternate between two closest valid frequencies. We now show the optimality of the alternation scheme by analyzing the power consumption of one of the \(N\) functional units.
We prove a stronger version of optimality for the alternation scheme. Instead of considering a machine with a number of discrete frequencies, we assume that the machine has only a range of invalid frequencies. To be exact, given two frequencies, \( g_1 \) and \( g_2 \) (\( 0 \leq g_1 < g_2 \)), the machine can operate at any frequency \( f \), except when \( g_1 < f < g_2 \). When the optimal frequency lies between \( g_1 \) and \( g_2 \), the simplified constraint is the same as the original constraint to the alternation scheme. However, the new constraint allows for all other execution schemes, not just those of a fixed set of valid frequencies. It should also be mentioned that the lowest valid frequency on any machine is 0. In addition, our program and machine model (Section 2) does not permit a program to run on a frequency that is higher than the highest machine frequency. In other words, it is not possible for the optimal frequency to lie outside valid frequencies. Thus the simplified frequency constraint covers all cases of discrete frequencies and does not lose any generality.

We now formulate the problem. Given a machine with a range of invalid frequencies between a lower point \( g_1 \) and a higher point \( g_2 \) (although \( g_1 \) and \( g_2 \) are valid). Assume a program with \( N \) blocks, each has \( a_i \) operations for the functional unit and takes \( t_i \) to execute. The operating frequency for each block is \( f_i = \frac{a_i}{t_i} \), which is valid, i.e. outside the range of \( g_1 \) and \( g_2 \). From Theorem 2, the optimal frequency, \( f \), is \( \frac{\sum_{i=1}^{n} a_i}{\sum_{i=1}^{n} t_i} \). Now suppose that the optimal frequency is not a valid frequency, that is, \( f > g_1 \) and \( f < g_2 \).

The alternation scheme runs the unit by frequency \( g_1 \) in time \( T_1 \) and by \( g_2 \) in \( T_2 \), where \( T_1 + T_2 = \sum_{i=1}^{n} t_i \) and \( g_1 T_1 + g_2 T_2 = \sum_{i=1}^{n} a_i \). The energy consumption of the original scheme, \( E_{original} \), is \( \sum_{i=1}^{n} f_i t_i \). The energy consumption of the alternation scheme, \( E_{opt} \), is \( g_1^2 T_1 + g_2^2 T_2 \). The following theorem states that the alternation scheme consumes the least amount of energy, given the constraints on valid frequencies.

**Theorem 3** (Theorem for discrete frequencies): \( \sum_{i=1}^{n} f_i t_i \geq g_1^2 T_1 + g_2^2 T_2 \), where \( f_i, g_1, g_2 \geq 0 \), \( t_i > 0 \), \( T_1 + T_2 = \sum_{i=1}^{n} t_i \), \( \sum_{i=1}^{n} f_i t_i = g_1 T_1 + g_2 T_2 \), and either \( f_i \geq g_2 \) or \( f_i \leq g_1 \).

Like the proof of Theorem 1, we use induction on \( n \), and we can again reduce the case of \( n = n + 1 \) to the case of \( n = 2 \), which is equivalent to the following lemma.

**Lemma 4.2** \( f_1^2 t_1 + f_2^2 (T - t_1) \geq g_1^2 t_2 + g_2^2 (T - t_2) \), where \( 0 \leq f_1 \leq g_1 < g_2 \leq f_2 \), \( t_1, t_2 > 0 \), and \( f_1 t_1 + f_2 (T - t_1) = g_1 t_2 + g_2 (T - t_2) \).

**Proof:** Let \( W = f_1 t_1 + f_2 (T - t_1) \). We can represent \( t_1 \) and \( t_2 \) with \( W \), that is, \( t_1 = \frac{W - g_2 T}{f_1 - f_2} \) and \( t_2 = \frac{W - g_1 T}{g_1 - g_2} \). In addition, let \( f'T = W \). Substitute \( t_1 \) and \( t_2 \) with \( f' \) in the inequality and simplify the equation give us the following inequality, which is surprisingly well behaved considering that it has five variables that are only loosely constrained.

\[
ff_1^2 + f_2(f - f_1)(f_1 + f_2) \geq g_1^2 + g_2(f - g_1)(g_1 + g_2)
\]

where \( f_1 \leq g_1 \leq f \leq g_2 \leq f_2 \).

Since \( f_1 \geq g_1 \) and \( f_2 \geq g_2 \), it is sufficient to show \((f - f_1)(f_1 + f_2) \geq (f - g_1)(g_1 + g_2)\). Define a function \( F(x) = (f - x)(x + g_2) = -x^2 + (f - g_2)x + f g_2 \). \( F(x) \) hits its maximal point at \( x = \frac{f - g_2}{2} \). Since \( f \leq g_2 \), \( F(x) \) is decreasing when \( x \geq 0 \). Hence, \( F(f_1) \geq F(g_1) \) when \( f_1 \leq g_1 \). Thus, the inequality holds, so is the lemma and Theorem 3.

We make two additional comments on the proof. First, the solution for discrete frequencies can be generalized to multiple functional units by applying the alternation scheme on each unit. Second, this alternation scheme is best among all execution schemes that require the same or less execution time.

In essence, Lemma 4.2 is a constrained version of Lemma 3.1. These two lemmas form the basis for the whole proof system of the paper. They establish the optimality property for a single functional unit in two execution cycles. The rest of the proof extends them to multiple functional units and time cycles.
5 Energy Consumption and Program Performance

Our theorems specify the optimal program organization for minimal energy consumption. The optimal order for energy is substantially different from that for performance. We illustrate this difference with an example.

The example in Figure 2 shows the different reorderings needed for energy and performance. Here we assume a machine with one integer unit and one floating-point unit, with the same maximal frequency $f$. The execution time of Part (a) is no less than $\frac{9}{f}$. Part (b) shows a reordered program that runs faster—in time $\frac{8}{f}$. This order in fact offers the best performance because FPU must execute all 8 floating-point operations. However, the program is not most energy efficient because it has uneven instruction balances. The version in Part (c) has constant instruction balances and, according to our theorem, consumes minimal energy. Part (c) also yields optimal performance, $\frac{7}{f}$.

![Figure 2: Difference between performance and energy optimization](image)

The key issue in obtaining best performance is the use of critical resource. If the critical resource is fully utilized, the performance is optimal (without removing program instructions). In the previous example, the program has more floating-point operations than integer operations, so FPU is the critical resource. To keep the resource fully utilized, each instruction block must have no lower demand for the critical resource than for any other resource. In terms of instruction balance, the ratio of integer operations to floating-point operations must be smaller than or equal to one (assuming the same amount of machine resources). The first program is not performance optimal because the first block has a ratio of 1.5. Both the second and third program correct this problem and obtain the fastest speed. In fact, all reordering schemes that bound the balance of the two blocks to one can have the optimal performance.

The difference is now clear in terms of instruction balance. For best performance, we want full utilization of the critical resource, or a bounded balance in all instruction blocks. For minimal energy, we need a stronger condition, where all blocks have the same instruction balance. Table 2 summarizes the differences between performance and energy. It shows that program reordering for energy is a different problem that warrants further investigation.

<table>
<thead>
<tr>
<th></th>
<th>Improving Performance</th>
<th>Saving Energy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goal</td>
<td>full utilization of critical resource</td>
<td>balanced use of all resources</td>
</tr>
<tr>
<td>Reordering</td>
<td>bounded instruction balances</td>
<td>constant instruction balances</td>
</tr>
</tbody>
</table>

Table 2: Comparison of Program Reordering for Performance and for Energy
6 Related Work

In the past, power models at the CMOS level were studied for the processor core [2] and for memory hierarchy [7]. Instruction-level power consumption for fixed-configuration processors was measured for real machines [20, 18]. Recently, architectural-level simulators have been developed [22, 1]. These models are more detailed and closer to a real machine than ours. However, they did not consider fine-grain reconfigurable systems as we do here. We plan to extend our power model to real systems using previous modeling techniques.

A large body of work has been devoted to circuit or architectural improvement for power efficiency. Circuit-level features cannot be directly controlled by software, but architectural ones can. Our strategy depends on dynamic voltage and frequency scaling [2], which has been used by a commercial processor, Transmeta Crusoe [6]. Other architectural techniques include clock gating [11] and pipeline gating [16], which curb redundant circuit activities and are orthogonal to dynamic voltage scaling. New buffering techniques have been designed for memory subsystem, including filter cache [14] and block buffering [13]. Our goal is complementary, which is to reorganize programs to best exploit these hardware features.

Software techniques have been studied for reducing energy usage [20, 19, 21]. Tiwari et al. reported that the most energy saving was obtained by reducing memory misses (up to 40% saving) and the least effective was energy-based code generation and instruction scheduling [21]. Because better caching leads to significant power saving, it is not surprising that memory-hierarchy optimizations become popular on portable systems. Vijaykrishnan et al. evaluated a set of compiler optimizations [22]. These previous techniques focused on demand reduction, which does not tap into the deeper potential of reconfigurable hardware. We focus on demand reordering, which complements demand reduction.

Researchers also studied better software feedback for reconfigurable hardware [12] and OS support for paging [15] and disk scheduling [10]. These techniques do not change program demand and are orthogonal to software reorganization.

For higher performance, researchers have studied program reorganization for many years. Here we review the ones that are related to instruction balance. Balance was introduced to model FPU throughput and load/store bandwidth [4]. Transformations such as unroll-and-jam are used to improve program balance [3, 5]. The problem of memory bandwidth was studied later [17]. To consider all levels of memory hierarchy, our earlier work extended the definition of balance from a ratio to a tuple [8]. In this report, we further extend the definition to include all components of a computer including those within CPU.

7 Conclusion and Future Work

This paper has presented a theoretical result to an important optimization problem, which is to find the optimal program organization for energy consumption, considering discrete time cycles and machine frequencies. It has proved that a program with constant instruction balances consumes the least amount of energy on a machine with dynamic scaling, that balancing program instructions guarantees power saving without performance degradation, and that achieving minimal energy consumption requires a stronger version of reordering than obtaining best performance does.

We are currently measuring instruction balances in benchmark programs. More importantly, we are designing a Smooth compiler for improving instruction balances in large programs by building upon the global program and data transformations that we developed in previous work [9].
Acknowledgment

We would like to thank Dr. Bin Han for providing the idea that led to the proof of Lemma 3.1. We are also grateful to Xianghui Liu and the members of the system group at the Computer Science Department of University of Rochester for their helpful discussions.

References


