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

Technical Report UR-CS-TR-739

Abstract

A computer processor consists of multiple functional units that carry out different sets of instructions. At each time of execution, a program may have a varied number of instructions for each functional unit. Recent studies have exploited this difference to save energy. The idea is to slow functional units that have a lower load. As a result, the number of idle CPU cycles is reduced and so is the waste of power. Symmetrical to reconfiguring hardware is reorganizing software. For each program segment, we can alter its demands for different functional units by exchanging its instructions with those from other parts of the program. In other words, we can change the balance of instructions 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 a consistent instruction balance always consumes less power than the same program with an uneven balance. In addition, the paper examines the relation between energy consumption and program performance. It shows that as far as instruction balance is concerned, reducing power is a more complex problem than improving performance.

1 Introduction

As personal and network computing become ubiquitous, a large number of computers are and will be portable devices powered by batteries. For these devices, lower energy intake would allow for smaller batteries, lower device weight and longer uninterrupted operation. Therefore, managing the energy consumption of portable processors has become important, because reducing power consumption directly leads to lower cost and better service.

Electric power is a product of current and voltage. In the case of a computer, both factors are largely determined by its operating frequency, that is, the clock rate. Indeed, the power consumption is estimated to be a cubic function over frequency. Since not all parts of a computer are busy during a given time, we may lower the frequency of less busy parts to save power while maintaining the overall program speed. For example, when CPU is frequently idle due to slow memory loads, the processor can save power by operating at a lower frequency. This technique is called *dynamic frequency and voltage scaling* [1].

The frequency adjustment by hardware, however, is not a complete solution because it does not consider possible reorganizations in software. In particular, a compiler can change the program demand at each point of the computation. For example, a compiler may increase the CPU workload in one part of a program by gathering computations from other parts. Since program reordering leads to a different idle/busy pattern in hardware, an interesting question is whether program reordering can bring about additional power saving.

In this paper, we explore the theoretical lower bound of the energy consumption assuming that both a program and a machine are fully adjustable. We model a computer as a collection of functional units whose frequency can be independently adjusted. We model a program by its execution trace, which is a sequence of instructions each uses a single functional unit. For the purpose of illustration, we will assume a computer with two functional units: integer unit (ALU) and floating-point unit (FPU). However, the result can be generalized to a processor with any number of components, including cache and main memory.

Figure 1 illustrates the effect of program reordering on power saving. Part (a) shows an example program consisting of two blocks of instructions. The mix or the balance between floating-point operations (*fp op*) and integer operations (*int op*) is different in the two instruction blocks. Assuming these instructions can be arbitrarily reordered, we transform the program so that the instruction balance is the same between two blocks. The balanced program is shown in Part (b). The energy consumption for both programs is shown in Table 1.

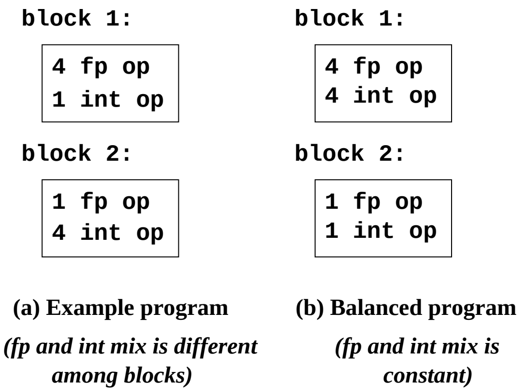


Figure 1: Example instruction balance

Table 1 shows three configurations: original program without and with frequency scaling, along with the balanced program with frequency scaling. Each configuration includes three rows: two for instruction blocks

version	block	frequency	exe. time (t)	Energy $E = tf^3c$
example program	block 1	$f_{fp} = f_{int} = f_{max}$	$\frac{4}{f_{max}}$	$8f_{max}^2 c$
with no	block 2	$f_{fp} = f_{int} = f_{max}$	$\frac{4}{f_{max}}$	$8f_{max}^2 c$
frequency scaling	total		$\frac{8}{f_{max}}$	$16f_{max}^2 c$
example program	block 1	$f_{fp} = f_{max}$,$f_{int} = \frac{1}{4}f_{max}$	$\frac{4}{f_{max}}$	$4\frac{1}{16}f_{max}^2 c$
with	block 2	$f_{fp} = \frac{1}{4}f_{max}$,$f_{int} = f_{max}$	$\frac{4}{f_{max}}$	$4\frac{1}{16}f_{max}^2 c$
frequency scaling	total		$\frac{8}{f_{max}}$	$8.13f_{max}^2 c$
balanced program	block 1	$f_{fp} = \frac{5}{8}f_{max}$,$f_{int} = \frac{5}{8}f_{max}$	$\frac{32}{5f_{max}}$	$3\frac{1}{8}f_{max}^2 c$
with	block 2	$f_{fp} = \frac{5}{8}f_{max}$,$f_{int} = \frac{5}{8}f_{max}$	$\frac{8}{5f_{max}}$	$\frac{25}{32}f_{max}^2 c$
frequency scaling	total		$\frac{8}{f_{max}}$	$3.91f_{max}^2 c$

Table 1: Power consumption of example and balanced program

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, f_{int} and f_{fp}. The fourth column shows the execution time, t, which is the number of operations divided by the operating frequency. The energy consumption, E, shown in the last column, is calculated as tf^3c, where c is a constant.

Without frequency adjustment, all units run at the peak speed, f_{max}. With frequency scaling, only one unit runs at the peak speed, the other unit runs at a lower frequency, $\frac{1}{4}f_{max}$. For the balanced program, both units run at a lower speed, $\frac{5}{8}f_{max}$. The three rows labeled with "total" shows the overall speed and energy consumption. All three configurations have the same execution time, $\frac{8}{f_{max}}$. The energy consumption, however, is significantly different. Frequency scaling consumes a little more than half of the energy (51%) compared to no scaling; program balancing further reduces the energy consumption by over a half, requiring only 24% of the initial energy. Thus, program reorganization can save substantially more energy than hardware frequency scaling alone.

The rest of this paper formally presents the idea used in the previous example. Section 2 defines the program and machine model and formulates the concept of instruction balance. Section 3 proves the central theorem of the paper, which characterizes the relation between instruction balance and energy consumption. Section 4 extends the proof to a more general machine model. Section 5 addresses the subtle distinction between energy consumption and program performance. Finally, Section 6 reviews related work and Section 7 concludes.

2 Program and Machine Model

Program Model The execution trace of 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 floating-point operations in the block. Section 4 will extend this definition to programs with more than two instruction types. Instruction blocks have two properties. The first is that different types of operations inside a block can be executed independently. In other words, the integer and floating-point units can execute under independent frequencies. The second property is that a processor must finish all instructions of one block before starting on the next block. In the extreme case, we can treat the set of instructions executed at each machine cycle as a block.

We define the instruction *balance* for a block B_i as the ratio $\dfrac{a_i}{b_i}$. 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 a *constant instruction balance*. We call such a program a *balanced program*. Note that the condition can be rewritten as $\frac{a_i}{a_j} = \frac{b_i}{b_j}$. The second format is more convenient when we later extend the formulation to instructions of more than two types. We also note that a constant balance is not necessarily a unit balance. 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 Power dissipation for CMOS circuits can be accurately modeled with sample equations. The dominant source of power dissipation in a digital CMOS is the dynamic power dissipation which can be expressed by $P_{dynamic} \approx V^2 \cdot f \cdot C$ where V is the supply voltage, f is the clock speed and C is the effective switching capacitance. Since in practice V varies approximately linearly with f, power consumption varies approximately linearly with f^3. The energy consumption is equal to the product of the power consumption and the execution time.

We make two assumptions on the operating frequency. First, the clock rate of different functional units can be independently adjusted. Second, the clock rate can be set to any non-negative rational number at any time. The second assumption will be relaxed later in Section 4.

3 Instruction Balance and Energy Consumption

This section proves the central theorem of the paper, which states that a constant instruction balance consumes minimal energy. The section first formulates the problem and then presents the formal proof.

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_1 with dynamic frequency scheduling can be computed in the following three steps.

- 1. The time needed for B_i : $\dfrac{M_i}{f}$ where $M_i = max(a_i, b_i)$, assuming $M_i > 0$.
- 2. The energy consumption for B_i: $\dfrac{M_i}{f}(f^3 + (\dfrac{m_i}{M_i}f)^3)$, where $m_i = min(a_i, b_i)$
- 3. The total energy consumption :

$$
\begin{aligned}
E_1 &= \sum_{i=1}^{n} \frac{M_i}{f}(f^3 + (\frac{m_i}{M_i}f)^3) \\
&= \sum_{i=1}^{n} M_i(f^2 + \frac{m_i^3}{M_i^3}f^2) \\
&= \sum_{i=1}^{n} M_i(1 + \frac{m_i^3}{M_i^3})f^2 \\
&= \sum_{i=1}^{n} \frac{M_i^3 + m_i^3}{M_i^2}f^2 \\
&= \sum_{i=1}^{n} \frac{a_i^3 + b_i^3}{M_i^2}f^2
\end{aligned}
$$

So we have $E_1 = \sum_{i=1}^{n} \dfrac{a_i^3 + b_i^3}{M_i^2} f^2$.

If we re-balance the program, i.e. transforming the program into $P' = (B_1', ..., B_i', ..., B_n'), B_i' = (a_i', b_i'), \frac{a_i'}{b_i'} = \frac{A}{B}$, $i = 1, ..., n$ where $A = \sum_{i=1}^n a_i$ and $B = \sum_{i=1}^n b_i$. Suppose we keep the same execution time in P' as in P, which is $\sum_{i=1}^{n} \frac{M_i}{f}$ The following three steps compute the energy consumption E_2 for the transformed program, P'.

- 1. The frequency of the integer unit: $\frac{A}{\frac{\sum_{i=1}^n M_i}{f}}$
- 2. The frequency needed to do the float operation: $\frac{B}{\frac{\sum_{i=1}^n M_i}{f}}$
- 3. The energy consumption:
$$\begin{aligned} E_2 &= \frac{\sum_{i=1}^n M_i}{f}\left(\left(\frac{A}{\frac{\sum_{i=1}^n M_i}{f}}\right)^3 + \left(\frac{B}{\frac{\sum_{i=1}^n M_i}{f}}\right)^3\right) \\ &= \frac{\sum_{i=1}^n M_i}{f}\left(\left(\frac{Af}{\sum_{i=1}^n M_i}\right)^3 + \left(\frac{Bf}{\sum_{i=1}^n M_i}\right)^3\right) \\ &= \frac{A^3 + B^3}{(\sum_{i=1}^n M_i)^2} f^2 \\ &= \frac{(\sum_{i=1}^n a_i)^3 + (\sum_{i=1}^n b_i)^3}{(\sum_{i=1}^n M_i)^2} f^2 \end{aligned}$$

Hence, we have $E_2 = \frac{(\sum_{i=1}^n a_i)^3 + (\sum_{i=1}^n b_i)^3}{(\sum_{i=1}^n M_i)^2} f^2$

To show that a balanced program consumes minimal power, we need to show that $E_2 \leq E_1$ for any program P. Observe that in our formulation E_2 and E_1 have common factors f^2, so we only need to show that

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

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 are non-negative integers and $M_i \geq max(a_i, b_i)$ (assuming $M_i > 0$). The equality holds when and only when $\frac{a_i}{b_i} = \frac{a_j}{b_j}$ for all i and j.

The inequality says that the energy consumption of any program, represented by the left-hand side formula, is always greater than the energy consumption of its balanced self, represented by the right-hand side formula, assuming the balanced program has the same execution time as the original program. In other words, this inequality says that instruction balancing can always save energy for any unbalanced program. Although less obvious, the theorem is a bit stronger than what is necessary here. It assumes $M_i \geq max(a_i, b_i)$ while in our formulation, $M_i = max(a_i, b_i)$. The reason for this augmentation will become clear when we extend the theorem to more than two instruction types in Section 4.

We prove the theorem by induction. The general idea is first to reduce the case of $n + 1$ to the case of $n = 2$ by induction hypothesis, then use calculus method to prove the inequality is true when $n = 2$.

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

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

We need to prove

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

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 only 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 $M^{'} \geq max(a^{'}, b^{'})$ since $M_i = max(a_i, b_i), i = 1, ..., n$. So we only 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 in general it is sufficient to prove that

$$\frac{a_1^3 + b_1^3}{M_1^2} + \frac{a_2^3 + b_2^3}{M_2^2} \geq \frac{(a_1 + a_2)^3 + (b_1 + b_2)^3}{(M_1 + M_2)^2}$$

where $M_1 \geq max(a_1, b_1), M_2 \geq max(a_2, b_2)$.

Lemma 3.1 $\frac{a_1^3 + b_1^3}{M_1^2} + \frac{a_2^3 + b_2^3}{M_2^2} \geq \frac{(a_1 + a_2)^3 + (b_1 + b_2)^3}{(M_1 + M_2)^2}$ *where* $M_1 \geq max(a_1, b_1), M_2 \geq max(a_2, b_2)$

Proof: For simplicity we can assume $M_2 = 1$. Otherwise we can do the following transformation:
$\frac{1}{M_2^2}(\frac{a_1^3 + b_1^3}{(\frac{M_1}{M_2})^2} + \frac{a_2^3 + b_2^3}{1^2}) \geq \frac{(a_1 + a_2)^3 + (b_1 + b_2)^3}{(\frac{M_1}{M_2} + 1)^2} \times \frac{1}{M_2^2}$

let $M_1 = \frac{M_1}{M_2}$, so we have

$\frac{a_1^3 + b_1^3}{M_1^2} + \frac{a_2^3 + b_2^3}{1^2} \geq \frac{(a_1 + a_2)^3 + (b_1 + b_2)^3}{(M_1 + 1)^2}$

So now let $M = M_1$, we prove that $\frac{a_1^3 + b_1^3}{M^2} + \frac{a_2^3 + b_2^3}{1^2} \geq \frac{(a_1 + a_2)^3 + (b_1 + b_2)^3}{(M + 1)^2}$

$\frac{a_1^3 + b_1^3}{M^2} + \frac{a_2^3 + b_2^3}{1^2} \geq \frac{(a_1 + a_2)^3 + (b_1 + b_2)^3}{(M + 1)^2}$
$\Longleftrightarrow (M + 1)^2(a_1^3 + b_1^3 + M^2(a_2^3 + b_2^3)) \geq M^2((a_1 + a_2)^3 + (b_1 + b_2)^3)$

$\Longleftrightarrow (M^2+2M+1)(a_1^3+b_1^3+M^2a_2^3+M^2b_2^3)$
$\geq M^2(a_1^3+a_2^3+b_1^3+b_2^3+3a_1a_2(a_1+a_2)+3b_1b_2(b_1+b_2))$
$\Longleftrightarrow \dfrac{(2M+1)(a_1^3+b_1^3)+(2M^3+M^4)(a_2^3+b_2^3)}{M^2} \geq 3a_1a_2(a_1+a_2)+3b_1b_2(b_1+b_2)$
$\Longleftrightarrow (2M^{-1}+M^{-2})a_1^3+(2M+M^2)a_2^3+(2M^{-1}+M^{-2})b_1^3+(2M+M^2)b_2^3 \geq 3a_1a_2(a_1+a_2)+3b_1b_2(b_1+b_2)$
Thus if we can prove that
$(2M^{-1}+M^{-2})a_1^3+(2M+M^2)a_2^3 \geq 3a_1a_2(a_1+a_2)$
$(2M^{-1}+M^{-2})b_1^3+(2M+M^2)b_2^3 \geq 3b_1b_2(b_1+b_2)$,
then we are done. Also since the above two inequalities are the same, so we only need to prove one of them. Now we prove

$$(2M^{-1}+M^{-2})a_1^3+(2M+M^2)a_2^3 \geq 3a_1a_2(a_1+a_2).$$

Lemma 3.2 $(2M^{-1}+M^{-2})a_1^3+(2M+M^2)a_2^3 \geq 3a_1a_2(a_1+a_2)$ *where* $M, a_1, a_2 > 0$.

Proof: Define a function $f(M) = (2M^{-1}+M^{-2})a_1^3+(2M+M^2)a_2^3, M, a_1, a_2 > 0$.
Note that $f(\frac{a_1}{a_2}) = 3a_1a_2(a_1+a_2)$. We claim that the function $f(M), M > 0$ achieve its minimum value at $\frac{a_1}{a_2}$, i.e.
$f(M) \geq f(\frac{a_1}{a_2}) = 3a_1a_2(a_1+a_2)$.
The derivative $f^{'}(M) = (-2M^{-2}-2M^{-3})a_1^3+(2+2M)a_2^3$.
The second derivative $f^{''}(M) = (4M^{-3}+6M^{-4})a_1^3+2a_2^3 > 0$.
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 $f^{'}(\frac{a_1}{a_2}) = 0$.
Since $f^{'}(M)$ is increasing, hence $f^{'}(M) \leq 0$ on $(0, \frac{a_1}{a_2}]$ and $f^{'}(M) \geq 0$ on $[\frac{a_1}{a_2}, \infty)$.
$f^{'}(M) \leq 0$ on $(0, \frac{a_1}{a_2}]$, so $f(M) \geq f(\frac{a_1}{a_2}), \forall M \in (0, \frac{a_1}{a_2}]$.
$f^{'}(M) \geq 0$ on $[\frac{a_1}{a_2}, \infty)$, so $f(M) \geq f(\frac{a_1}{a_2}), \forall M \in [\frac{a_1}{a_2}, \infty)$.
Therefore, $f(M) \geq f(\frac{a_1}{a_2}), \forall M > 0$.
So finally we have $f(M) = (2M^{-1}+M^{-2})a_1^3+(2M+M^2)a_2^3 \geq 3a_1a_2(a_1+a_2)$.

Now apply the Lemma 3.2, we have:
$(2M^{-1}+M^{-2})a_1^3+(2M+M^2)a_2^3 \geq 3a_1a_2(a_1+a_2)$.
$(2M^{-1}+M^{-2})b_1^3+(2M+M^2)b_2^3 \geq 3b_1b_2(b_1+b_2)$.

Therefore Lemma 3.1 holds. The theorem then follows from Lemma 3.1.

4 Extensions to the Machine Model

The basic theorem just proved assumes a machine with two functional units that operate on a frequency of any rational non-negative number. This section removes these two restrictions. It augment the theorem for 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 a constant instruction if* $\dfrac{a_{i1}}{a_{j1}} = \cdots = \dfrac{a_{im}}{a_{jm}}, \forall B_i, B_j$.

Theorem 2 *(Generalization of Theorem 1):*

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

where $a_{i1}, ..., a_{im}$ *are non negative integers and* $M_i \geq max(a_{i1}, ..., a_{im})$ *(assuming* $M_i > 0$*).*

We can proof the generalized theorem by induction on m. If $m = 2$, theorem 1 applies and clearly the new theorem holds. 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 $M_i \geq max\{a_{i1}, ..., a_{im}, a_{im+1}\} \geq max\{a_{i1}, ..., a_{im}\}$, so by induction hypothesis we have

$$\sum_{i=1}^{n} \frac{a_{i1}^3 + a_{i2}^3 + \cdots + a_{im}^3 + a_{im+1}^3}{M_i^2}$$
$$= \sum_{i=1}^{n} \frac{a_{i1}^3 + a_{i2}^3 + \cdots + a_{im}^3}{M_i^2} + \sum_{i=1}^{n} \frac{a_{im}^3}{M_i^2}$$
$$\geq \frac{(\sum_{i=1}^{n} a_{i1})^3 + (\sum_{i=1}^{n} a_{i2})^3 + \cdots + (\sum_{i=1}^{n} a_{im})^3}{(\sum_{i=1}^{n} M_i)^2} + \sum_{i=1}^{n} \frac{a_{im}^3}{M_i^2}.$$

Note that $\sum_{i=1}^{n} \frac{a_{im+1}^3}{M_i^2} = \sum_{i=1}^{n} \frac{a_{im+1}^3 + b_{im+1}^3}{M_i^2}$ where $b_{im+1} = 0, \forall i = 1, ..., n$, and
$M_i \geq max\{a_{i1}, ..., a_{im}, a_{im+1}\} \geq max\{a_{im+1}\} \geq max\{a_{im+1}, b_{im+1}\}$,
so by theorem 1, we get

$$\sum_{i=1}^{n} \frac{a_{im+1}^3 + b_{im+1}^3}{M_i^2} \geq \sum_{i=1}^{n} \frac{a_{im+1}^3}{M_i^2} + \sum_{i=1}^{n} \frac{b_{im+1}^3}{M_i^2} = \frac{(\sum_{i=1}^{n} a_{im+1})^3}{(\sum_{i=1}^{n} M_i)^2}.$$

Hence we have

$$\sum_{i=1}^{n} \frac{a_{i1}^3 + a_{i2}^3 + \cdots + a_{im}^3 + a_{im+1}^3}{M_i^2}$$
$$\geq \frac{(\sum_{i=1}^{n} a_{i1})^3 + (\sum_{i=1}^{n} a_{i2})^3 + \cdots + (\sum_{i=1}^{n} a_{im})^3}{(\sum_{i=1}^{n} M_i)^2} + \sum_{i=1}^{n} \frac{a_{im}^3}{M_i^2}$$
$$\geq \frac{(\sum_{i=1}^{n} a_{i1})^3 + (\sum_{i=1}^{n} a_{i2})^3 + \cdots + (\sum_{i=1}^{n} a_{im})^3}{(\sum_{i=1}^{n} M_i)^2} + \frac{(\sum_{i=1}^{n} a_{im+1})^3}{(\sum_{i=1}^{n} M_i)^2}$$

$$= \frac{(\sum_{i=1}^{n} a_{i1})^3 + (\sum_{i=1}^{n} a_{i2})^3 + \cdots + (\sum_{i=1}^{n} a_{im+1})^3}{(\sum_{i=1}^{n} M_i)^2}.$$

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 an uneven instruction balance, 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. To be exact, assume that a functional unit has k valid non-negative frequencies from lowest to highest, f_1, f_2, ..., f_k, and that the optimal frequency is f, and the total execution time is T. In addition, assume that f is not a valid frequency, that is, $f > f_i$ and $f < f_{i+1}$. The alternation scheme runs the unit by frequency f_i in time t_1 and by f_{i+1} in t_2, where $t_1 + t_2 = T$ and $f_i t_1 + f_{i+1} t_2 = fT$. A formal proof can show that the alternation scheme incurs minimal additional energy compared to running the functional unit by the optimal frequency. Any other execution scheme consumes more energy than the alternation scheme. To save space, we present an informal description of the proof instead of the full proof.

If the functional unit uses a valid frequency that is not f_i or f_{i+1}, then it must use a frequency lower than f_i and a frequency higher than f_{i+1}. Assuming that the amount of work lost due to the lower frequency is w, the same amount of work must be compensated at a higher frequency. Since the change in energy consumption is a cubic function of w, the derivative of that function is a square function of w. Therefore, the increased energy consumption is larger when w is not zero. In other words, any use of a frequency other than f_i and f_{i+1} would consume more energy. Thus, alternating between the two closest valid frequencies is most energy efficient.

Finally, we make two points. 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. The same is true for all earlier versions of the theorem. However, performance directly impacts energy consumption. The next section examines the relation between energy consumption and program performance.

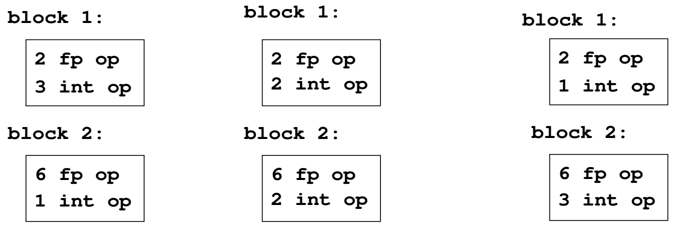
5 Energy Consumption and Program Performance

Energy consumption and program performance are related but different issues. This section analyzes this difference first in terms of their problems and then in terms of their solutions.

5.1 Problem of Energy and Performance

Energy consumption and program performance are directly related. One can always trade performance for more power saving by running the program at a lower clock rate. The relation, however, becomes less clear once program reordering is allowed. The same kind of reordering may benefit both program performance and energy consumption. The real question is whether the problem of energy consumption requires substantially different reordering than the problem of performance. The answer is yes. We first illustrate the difference by an example in Figure 2.

Part (a) of Figure 2 is a program with two blocks. Assuming a maximal clock rate of f, the fastest execution time is $\frac{9}{f}$. Part (b) shows a reordered program that runs faster—in time $\frac{8}{f}$. This order in fact



(a) Example program *(unbalanced, finishing 9 ops in full speed)*

(b) Performance optimized *(unbalanced, finishing 8 ops in full speed)*

(c) Energy optimized *(balanced, finishing 8 ops in full speed)*

Figure 2: Difference between performance and energy optimization

offers the best performance because FPU must execute all 8 floating-point operations. The two blocks in Part (b), however, do not have the same balance. A third reordering, a balanced program, is shown in Part (c). In terms of performance, the third program is no better than the second, although both offer the best performance. However, in terms of energy consumption, the third program is always more efficient than the second if they run at the same speed. Therefore, the problem of energy and performance is different because the best program organization for performance may not be the best choice for energy consumption.

The key issue in obtaining best performance is the use of critical resource. If the critical resource is fully utilized through the whole execution, no further performance gain would be possible (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 higher or same 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 are performance optimal. In fact, all reordering schemes that bound the balance of the two blocks to one are performance optimal.

Now the distinction can be exactly described in terms of instruction balance. For best performance, we want bounded balance in all instruction blocks, where the demand for the critical resource is no lower than the demand for other resources. For minimal energy, we need consistent or constant balance, where the relative demands for all resources are identical among instruction blocks. Given a program, it has exactly one program organization that has a constant balance but it can have many organization schemes that have a bounded balance. The constant balance must be also a bounded balance, but a bounded balance may not be the constant balance. Hence, the energy problem is deeper than the performance problem, and finding the solution for the energy problem is harder than finding a solution for the performance problem.

5.2 Performance Improvement and Power Reduction

For many years, researchers have studied program transformations for improving performance. The energy problem brings in two new elements. The first is the different goal of optimization; the second is that the target machine is no longer fixed and can be adjusted during execution. This section examines previous optimization approaches under these new conditions.

For machines with a RISC-like instruction set, previous compiler optimizations can be roughly categorized in two classes: *instruction reduction* and *instruction reordering*. The examples of the first class include classic code optimizations such as common-subexpression elimination and constant propagation. In recent years as the bottleneck shifted from CPU speed to memory speed, many techniques have been developed to alleviate the load on memory hierarchy. Examples include register allocation and cache blocking. They can be viewed as reduction techniques because they reduce the number of cache or memory accesses, although the reduction in cache misses do not change program instructions.

Instruction reduction saves energy just in the same way as it improves performance. While the fastest instructions are those that do not exist, these instructions are also the ones that consume the least energy. Therefore, existing techniques for instruction reduction work equally well in saving energy.

The second class of techniques that have been studied before is instruction reordering. Many reordering techniques are developed to enable instruction reduction. For example, loop transformations are used to enable better register allocation or cache utilization. Some reordering techniques, however, relies completely on the reordering to improve performance. Instruction scheduling, for example, reorders instructions to allow more operations to be executed in parallel.

Existing techniques for instruction reordering need to be re-targeted in order for them to effectively save energy. The theorem in this paper provides an theoretical proof on the optimal order of instructions. Reordering should be used to effect such an order, i.e. constant instruction balance. This is a more difficult task than performance improvement, where a bounded instruction balance suffices.

6 Related Work

Energy consumption has attracted a significant amount of attention recently. On the hardware side, researchers have explored dynamic techniques that change hardware configuration at runtime. Burd and Brodersen used dynamic frequency and subsequently voltage scaling as an effective solution for reducing power consumption [1]. Alternatively, a processor may completely stop the CPU when there is no work to do. The latter idea is called *clock gating* and was examined by Manne et al. for reducing the energy cost of branch misprediction [8]. Since these techniques modify only machine hardware, they complement our work, which is aimed at the best program organization on configurable hardware.

On the software side, researchers have studied program reorganization for performance reasons for years. Callahan et al. modeled program demand and machine resource with the concept of balance [3]. They considered the balance between FPU throughput and load/store bandwidth in single loop nests. For loops with too much memory operations, Callahan, Carr and Kennedy developed a compiler strategy that bring program balance closer to machine balance by reducing the number of memory operations [2, 4]. McCalpin observed that memory bandwidth is an important factor in determining program performance [9]. To consider all levels of memory hierarchy, Ding and Kennedy changed a balance from a ratio to a tuple that include load/store, cache and memory bandwidth along with FPU throughput [6]. Their experiment showed that memory bandwidth is the bottleneck among other resources for a set of scientific programs. Later, Ding's dissertation described a compiler strategy that reduces direct memory transfers by reorganizing global loops and arrays [5]. Ding's techniques have a global scope but they are aimed at the critical resource rather than the balance of all resources. These earlier work on software reorganization addressed only performance, not power consumption. They assumed fixed rather than adjustable hardware. They were targeted at instruction reduction, not instruction reordering. In this work, we address a broader optimization problem where both program and machine balance are adjustable. We provide a rigorous formulation and proof on the optimal program reordering.

Recently, Hsu et al. proposed compiler support for frequency and voltage scaling[7]. Their technique was to measure the program demand through a careful performance model based on CPU-memory balance. Their model considered compiler optimizations that may change the balance of individual loop nests. Their performance model can be used to efficiently control the reconfigurable hardware for both unoptimized or optimized programs.

7 Conclusion and Future Work

This paper has presented a theoretical solution to an important optimization problem, which is to find the optimal program organization for energy consumption, with no performance degradation and guaranteed power saving. It has proved that a constant instruction balance, combined with frequency and voltage scaling in hardware, consumes the least amount of energy. It also shown that achieving minimal energy consumption requires stronger reordering than obtaining best performance does.

We are currently measuring the instruction balance in benchmark programs and designing a global loop fusion algorithm that can smooth the instruction balance of large programs.

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 discussion and help that improved the presentation of this paper tremendously.

References

[1] T. Burd and R. Brodersen. Processor design for portable systems. *Journal of LSI Signal Processing*, 13(2-3):203–222, 1996.

[2] D. Callahan, S. Carr, and K. Kennedy. Improving register allocation for subscripted variables. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, White Plains, NY, June 1990.

[3] D. Callahan, J. Cocke, and K. Kennedy. Estimating interlock and improving balance for pipelined machines. *Journal of Parallel and Distributed Computing*, 5(4):334–358, August 1988.

[4] S. Carr and K. Kennedy. Improving the ratio of memory operations to floating-point operations in loops. *ACM Transactions on Programming Languages and Systems*, 16(6):1768–1810, 1994.

[5] C. Ding. Improving effective bandwidth through compiler enhancement of global and dynamic cache reuse. Ph.D Dissertation, Rice University, January 2000.

[6] C. Ding and K. Kennedy. Memory bandwidth bottleneck and its amelioration by a compiler. In *Proceedings of 2000 International Parallel and Distribute Processing Symposium (IPDPS)*, Cancun, Mexico, May 2000.

[7] C.-H. Hsu, U. Kremer, and M. Hsiao. Compiler-directed dynamic frequency and voltage scaling. In *Workshop on Power-Aware Computer Systems*, Cambridge, MA, 2000.

[8] S. Manne, A. Klauser, and D. Grunwald. Pipeline gating: speculation control for energy reduction. In *25th Annual International Symposium on Computer Architecture*, Barcelona, Spain, June 1998.

[9] J. D. McCalpin. Machine balance: Sustainable bandwidth vs. peak mflops. http://reality.sgi.com/mccalpin_asd/Balance.gif, 1998.