Instruction Balance and its Relation to Program Energy Consumption

Tao Li Chen Ding

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

Abstract. A computer 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 development has exploited this imbalance to save energy by slowing the components that have a lower load. Example techniques include dynamic scaling and clock gating used in processors from Transmeta and Intel. Symmetrical to reconfiguring hardware is reorganizing software. We can alter program demand for different components by reordering program instructions. This paper explores the theoretical lower bound of energy consumption assuming that both a program and a machine are fully adjustable. It shows that a program with a balanced load always consumes less energy than the same program with uneven loads under the same execution speed. In addition, the paper examines the relation between energy consumption and program performance. It shows that reducing power is a different problem than that of improving performance. Finally, the paper presents empirical evidence showing that a program may be transformed to have a balanced demand in most parts of its execution.

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 uninterrupted operation. As a result, managing the energy consumption of portable processors has become important because it directly leads to lower cost and better service. In addition, power reduction also helps traditional computing platforms such as desktop PCs and even supercomputers by reducing their 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 waste energy. The recent interest in energy efficiency has prompted rapid development of reconfigurable processors, which adjust hardware speed to match the dynamic demand of applications. For example, when CPU is under-utilized due to slow memory, its frequency and voltage are switched down to save energy. This technique is called *dynamic scaling*. It has already found its way into commercial processors such as Transmeta Crusoe and Intel XScale. For example, during execution, Crusoe TM5800 can switch its voltage between 0.9 volt to 1.3 volt and adjust its frequency between 367MHz and 800MHz. Given the improvement in hardware, a natural subsequent question is whether software can be adapted to fully utilize the emerging reconfigurable processors.

For many years, programs have been analyzed and optimized for performance. One effective optimization is demand reduction, which eliminates 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, it does not specifically utilize the adaptiveness of hardware. The question remains open on whether we can save additional energy after demand reduction has been applied. The following example will demonstrate that such opportunity exists. In particular, it will show that *demand reordering* can save energy on reconfigurable processors such as Transmeta Crusoe and Intel XScale.

A Motivating Example

For this example, we assume a simple machine with two components: CPU and main memory. We model each program as a sequence of instruction blocks, where blocks must be executed sequentially but operations within each block are fully parallel. We call the ratio of CPU operations (*cpu op*) to memory operations (*mem op*) the *instruction balance*. Part (a) of Figure 1 shows a program with two blocks with instruction balances 4 and $\frac{1}{4}$. Part (b) shows a reordered program that has an identical instruction balance of 1.

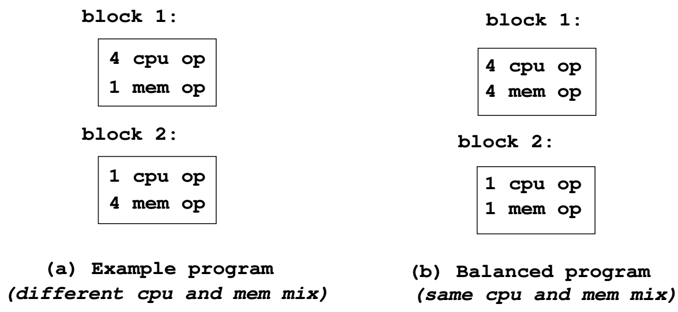


Fig. 1. Example of program reordering

Program	Block	Frequency	Exe. time (t)	Energy $E \approx tf^3c$
Example program with no dynamic scaling	block 1	$f_{cpu} = f_{mem} = f_{max}$	$\frac{4}{f_{max}}$	$8f_{max}^2c$
	block 2	$f_{cpu} = f_{mem} = f_{max}$	$\frac{4}{f_{max}}$	$8f_{max}^2c$
	total		$\frac{8}{f_{max}}$	$16f_{max}^2c$
Example program with dynamic scaling	block 1	$f_{cpu} = f_{max}$,$f_{mem} = \frac{1}{4}f_{max}$	$\frac{4}{f_{max}}$	$4\frac{1}{16}f_{max}^2c$
	block 2	$f_{cpu} = \frac{1}{4}f_{max}$,$f_{mem} = f_{max}$	$\frac{4}{f_{max}}$	$4\frac{1}{16}f_{max}^2c$
	total		$\frac{8}{f_{max}}$	$8.13f_{max}^2c$
Balanced program with dynamic scaling	block 1	$f_{cpu} = \frac{5}{8}f_{max}$,$f_{mem} = \frac{5}{8}f_{max}$	$\frac{32}{5f_{max}}$	$3\frac{1}{8}f_{max}^2c$
	block 2	$f_{cpu} = \frac{5}{8}f_{max}$,$f_{mem} = \frac{5}{8}f_{max}$	$\frac{8}{5f_{max}}$	$\frac{25}{32}f_{max}^2c$
	total		$\frac{8}{f_{max}}$	$3.91f_{max}^2c$

Table 1. Power consumption of example programs

Table 1 shows energy consumption for three configurations. The first is original program without dynamic scaling, where both hardware and software are fixed. The second is original program with dynamic scaling, where software is fixed but hardware is adaptive. The last is the balanced program with dynamic scaling, where software is reorganized and hardware is adaptive. Each configuration includes three rows: two for instruction blocks and one for their total. The data for each configuration are listed in columns. The third column lists the frequency used by CPU and memory unit, f_{cpu} and f_{mem}. 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 tf^3c, where t is the time, f is the frequency, and c is a architectural dependent constant. We assume the same constant for CPU and memory.

In the first configuration, all units run at the peak speed, f_{max}. In the second configuration, only one unit runs at the peak speed, the other unit runs at a lower frequency, $\frac{1}{4}f_{max}$. In the third configuration, both units run at a lower speed, $\frac{5}{8}f_{max}$. The three rows labeled with "total" give the overall speed and energy consumption. All three configurations have the same execution time, $\frac{8}{f_{max}}$. The energy consumption, however, is very different. Dynamic scaling saves about half of the energy (49%) compared to no scaling. Program reordering further reduces the energy consumption by over a half (another 52%). In other words, program reordering is able to double the energy saving without removing any instruction from the program.

The rest of this paper presents a theoretical basis for program reordering for the purpose of energy reduction. Section 2 defines the program and machine model. Section 3 and 4 prove the basic and extended theorems. Section 5 addresses the relation with performance optimization. Section 6 evaluates a benchmark program. Finally, Section 7 discusses related work, and Section 8 concludes.

2 Program and Machine Model

This section describes our 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 assume a machine with only two identical components running at any non-negative frequency. The extended theorems that include N asymmetrical components and discrete frequencies are given in Section 4.

Program Model We view a program, P, by its execution trace, which we model as 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 CPU operations, and b_i is the number of memory operations. We assume a sequential execution of blocks but no dependence among different types of operations inside a block. For example, we assume that CPU and memory operations can be executed independent of each other. This program model is not as unreasonable as it seems. For example, we can view 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 a constant instruction balance. We call the program a *balanced program.* The condition can be rewritten as $\frac{a_i}{a_j} = \frac{b_i}{b_j}$. The second format is more convenient when we extend the formulation to instructions of more than two types in Section 4. We note that a constant balance is not necessarily a unit balance. The number of CPU and memory operations does 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 multiple functional units whose frequency can be independently adjusted by hardware to match software demand (i.e. dynamic scaling). The total energy cost is the power consumption of each unit multiplied by the execution time. Burd and Brodersen divided CMOS power into static and dynamic dissipation [2]. 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, $P_i \approx f_i^3 c_i$, where c_i is an architectural dependent constant. We assume that all c_is are identical in proving the basic theorem. The extend theorems will remove this assumption in Section 4.2. We do not explicitly consider any overhead incurred by dynamic scaling. However, switching overhead does not change our theorem because in the optimal case, program demand is constant and incurs no switching cost.

The above machine and program model is too simple to model a real system. Program instructions cannot be arbitrarily reordered. The scaling between voltage and frequency is not linear [10]. However, the simplified model is useful in our 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: a program organization is optimal for energy if and only if all instruction blocks have the same instruction balance. To prove, we show that for any program with uneven instruction balances, its balanced counterpart can finish execution in the same amount of time but consume less energy. We next formulate the energy consumption of a program and its balanced counterpart.

3.1 Problem Formulation

Assume a program of n instruction blocks, $P = (B_1, ..., B_i, ..., B_n), B_i = (a_i, b_i)$, $i = 1, ..., n$. Let f be the maximum processor frequency. The original energy consumption $E_{original}$ is computed in the following three steps.

- The execution time of each block is $t_i = \dfrac{M_i}{f}$, where $M_i = max(a_i, b_i)$, and f is the maximal frequency of functional units.
- The power consumption of each block is $P_i = f^3 + (\dfrac{m_i}{M_i} f)^3$, where $m_i = min(a_i, b_i)$ and $M_i = max(a_i, b_i)$. The second term includes the effect of dynamic scaling.
- The energy consumption of the program is $$E_{original} = \sum_{i=1}^{n} t_i P_i = \sum_{i=1}^{n} \frac{M_i}{f}(f^3 + (\frac{m_i}{M_i} f)^3) = \sum_{i=1}^{n} \frac{M_i^3 + m_i^3}{M_i^2} f^2 = \sum_{i=1}^{n} \frac{a_i^3 + b_i^3}{M_i^2} f^2$$

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

- The frequency of CPU is $f_{cpu} = \frac{A}{t_{total}}$
- The frequency of the memory unit is $f_{mem} = \frac{B}{t_{total}}$
- The energy consumption is $$E_{balanced} = t_{total}(f_{cpu}^3 + f_{mem}^3) = \frac{A^3 + B^3}{(\sum_{i=1}^{n} M_i)^2} f^2 = \frac{(\sum_{i=1}^{n} a_i)^3 + (\sum_{i=1}^{n} b_i)^3}{(\sum_{i=1}^{n} M_i)^2} f^2$$

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

3.2 The Basic Theorem

The following 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 the same execution time. Therefore, program re-balancing saves energy. The theorem is stronger than required by the formulation. It allows for $M_i \geq max(a_i, b_i)$, while the formulation needs only $M_i = max(a_i, b_i)$. The generalization turns out to be the key to our proof. It makes the induction possible.

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.

Proof: We first use induction and then reduce the case of $n+1$ to the case of $n = 2$, which we prove with a calculus method.

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

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

We need to prove

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

After separating the terms involved in the case of n and applying the induction hypothesis, the inequality becomes

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

Now let $a' = \sum_{i=1}^{n} a_i, b' = \sum_{i=1}^{n} b_i, M' = \sum_{i=1}^{n} M_i$. We have $M' \geq max(a', b')$ because $M_i \geq max(a_i, b_i), i = 1, ..., n$. So we arrive at the same inequality as the case $n = 2$, which is

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

where $M_1 \geq max(a_1, b_1), M_2 \geq max(a_2, b_2)$. Next, we split the inequality into two parts: $\frac{a_1^3}{M_1^2} + \frac{a_2^3}{M_2^2} \geq \frac{(a_1 + a_2)^3}{(M_1 + M_2)^2}$ and $\frac{b_1^3}{M_1^2} + \frac{b_2^3}{M_2^2} \geq \frac{(b_1 + b_2)^3}{(M_1 + M_2)^2}$

Since the above two inequalities are equivalent, we prove the first one in the following lemma. Note that the lemma is stronger than required: M_1 and M_2 can be any positive number and do not have to be greater than a_1 and a_2.

Lemma 1. $\dfrac{a_1^3}{M_1^2} + \dfrac{a_2^3}{M_2^2} \geq \dfrac{(a_1 + a_2)^3}{(M_1 + M_2)^2}$, *where* $a_1, a_2, \geq 0$ *and* $M_1, M_2 > 0$.

Proof: We first convert M_2 to 1 by dividing both sides with M_2^2. Then we multiply both sides with the product of their denominators. The inequality is converted to

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

where $a_1, a_2, b_1, b_2 \geq 0$, and $M > 0$. If $a_1 = 0$ or $a_2 = 0$, the inequality holds trivially. Now we assume that $a_1, a_2 > 0$. Define

$$f(M) = (2M^{-1} + M^{-2})a_1^3 + (2M + M^2)a_2^3$$

where $M, a_1, a_2 > 0$. We will show that $f(M)$ researches its minimum value at $\dfrac{a_1}{a_2}$, which is $f(\dfrac{a_1}{a_2}) = 3a_1a_2(a_1 + a_2)$. The first and second derivatives are
$f^{'}(M) = (-2M^{-2} - 2M^{-3})a_1^3 + (2 + 2M)a_2^3$, and
$f^{''}(M) = (4M^{-3} + 6M^{-4})a_1^3 + 2a_2^3 > 0$
Since $f^{''}(M) > 0$, so $f(M) \geq f(\dfrac{a_1}{a_2}), \forall M > 0$. The lemma holds and the theorem follows.

4 Extended Theorems

The basic theorem assumes a machine that has two components and operates on any non-negative frequency. This section extends the theorem to a machine that has more than two components and operates on a set of fixed frequencies. The set of frequencies needs not to be the same for each component, nor does the constant factor in the energy equation. In the case of clock gating, a component has two frequencies: one is the operating frequency and the other is zero.

4.1 Multiple Functional Units

We generalize first the definition of instruction balance and then the theorem.

Definition 1. *Assume 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 an m-tuple* $(a_{i1}, ..., a_{im})$. *A program is said to have a constant instruction balance if* $\dfrac{a_{i1}}{a_{j1}} = \cdots = \dfrac{a_{im}}{a_{jm}}$, *for any* B_i, B_j.

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

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

where $a_{i1}, ..., a_{im} \geq 0$ *and* $M_i \geq max(a_{i1}, ..., a_{im}) > 0$.

Proof: We prove the generalized theorem by induction on m. The inductive case is established by applying theorem 1.

Since $M_i \geq max\{a_{i1}, ..., a_{im}, a_{im+1}\} \geq max\{a_{i1}, ..., a_{im}\}$, by induction hypothesis we have

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

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

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

Hence we have

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

So the generalized theorem holds.

The generalized theorem says that on machines with any number of components, a balance program consumes less energy than its unbalanced counterpart, given the same execution time.

4.2 Discrete Operating Frequencies

So far we have assumed that a frequency can be any non-negative rational number. On real machines, a frequency must be an integer and *valid frequencies* are pre-determined. For example, Transmeta Crusoe has 32 steps in its range of operating frequency and voltage. A less obvious case is clock gating, where a component either runs in full speed or is shut down completely. This case is equivalent to having two valid frequencies. 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 proving it for each component.

Assuming that the optimal frequency, f_{opt}, lies between two (closest) valid frequencies g_1 and g_2, we will show that alternating between these two frequencies consumes less energy than any other scheme that uses frequencies outside the range of g_1 and g_2. In other words, running the program at any other frequency at any time would consume more energy. In our proof, we compare the optimal scheme with schemes that can use all frequencies outside g_1 and g_2, not just those of a fixed set of valid frequencies. In addition, our program and machine model (Section 2) does not permit a frequency that is higher than the maximal machine frequency. So the following proof covers all cases of discrete frequencies and does not lose any generality.

We now formulate the problem. Assume a component that can operate on any frequency except 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 and takes t_i to execute. The operating frequency for each block is $f_i = \frac{a_i}{t_i}$, which must lie outside g_1 and g_2. From Theorem 2, the optimal frequency, f, is $\frac{\sum_{i=1}^{i=n} a_i}{\sum_{i=1}^{i=n} t_i}$. Assume that the optimal frequency is not a valid frequency, i.e., $f > g_1$ and $f < g_2$. The alternation scheme runs the component by frequency g_1 in time T_1 and by g_2 in T_2, where $T_1+T_2 = \sum_{i=1}^{i=n} t_i$ and $g_1T_1+g_2T_2 = \sum_{i=1}^{i=n} a_i$. The energy consumption of the original scheme, $E_{original}$, is $\sum_{i=1}^{i=n} f_i^3 t_i$. The energy consumption of the alternation scheme, E_{opt}, is $g_1^3T_1 + g_2^3T_2$. The following theorem states that the alternation scheme consumes the least amount of energy.

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

As in the proof of Theorem 1, we use induction on n and reduce the case of $n = n + 1$ to the case of $n = 2$, which is equivalent to the following lemma.

Lemma 2. $f_1^3 t_1 + f_2^3(T - t_1) \geq g_1^3 t_2 + g_2^3(T - t_2)$, *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 $W = f_1t_1 + f_2(T - t_1)$. We can represent t_1 and t_2 with W, that is, $t_1 = \frac{W - f_2T}{f_1 - f_2}$ and $t_2 = \frac{W - g_2T}{g_1 - g_2}$. In addition, let $fT = W$. Substitute t_1 and t_2 with f in the inequality and simplify the equation to the following inequality, which is surprisingly well behaved considering that it has five variables that are only loosely constrained.

$$ff_1^2 + f_2(f - f_1)(f_1 + f_2) \geq fg_1^2 + g_2(f - g_1)(g_1 + g_2)$$

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 $(f - f_1)(f_1 + g_2) \geq (f - g_1)(g_1 + g_2)$. Define a function $F(x) = (f - x)(x + g_2) = -x^2 + (f - g_2)x + fg_2$. $F(x)$ 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 three additional comments on the proof. First, the solution for discrete frequencies can be generalized to multiple components by applying the alternation scheme on each unit. Components can be asymmetrical. The minimal energy consumption of the whole system is minimal if and only if the energy consumption of each component is minimal. Second, this alternation scheme is best among all execution schemes that require the same or less execution time. Finally, the optimality holds when considering the overhead of switching between valid frequencies. The optimal alternation scheme switches only once, which is optimal.

In essence, Lemma 2 is a constrained version of Lemma 1. These two lemmas form the basis for the entire proof of the paper. Intuitively, they establish the optimality condition for a single component in two execution cycles. The rest of the proof extends them to multiple components and time cycles.

5 Energy Consumption and Program Performance

Program reordering has been studied for improving program performance. However, the problem is a new one in the case of energy reduction because the optimal order for energy is different from that for performance. We show this difference with an example.

We assume a machine with one integer unit and one floating-point unit, with the same maximal frequency f. Figure 2 shows three versions of the same program: unoptimized, optimized for performance, optimized for energy and performance. The execution time of the unoptimized program is $\frac{9}{f}$. The second version runs faster—in time $\frac{8}{f}$, which is optimal because FPU must execute all 8 floating-point operations. However, the second version is not most energy efficient because it has uneven instruction balances. The third program has constant instruction balances and, according to our theorem, consumes minimal energy. The third program also yields optimal performance, $\frac{8}{f}$.

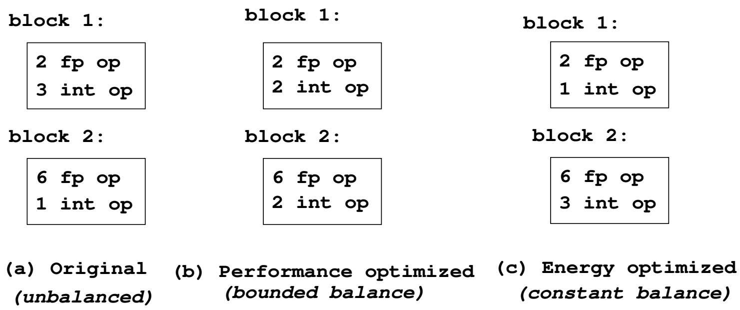


Fig. 2. Difference between performance and energy optimization

The simplified machine and program model does not consider the dependence and latency among program instructions. So the primary factor affecting performance is the utilization of critical resource. If the critical resource is fully utilized, the performance is optimal (assuming we cannot remove any instruction). 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, the ratio of floating-point operations to integer operations must be no less than one. The first program is not performance optimal because the first block has more integer operations than floating-point operations and therefore cannot fully utilize the critical resource. Both the second and third programs correct this problem and obtain the fastest speed. In fact, any reordering scheme is performance optimal if it bounds the balance to be no less than one.

The difference can now be described in terms of instruction balance. For best performance, we want full utilization of the critical resource and therefore a bounded balance. For minimal energy, we need a stronger condition that all blocks must have the same instruction balance. The requirement for a constant balance is even more important considering the switching cost of hardware. Constant balances do not require dynamic reconfiguration during execution. Table 2 summarizes the differences between issues of performance and those of energy.

	Improving Performance	**Saving Energy**
Goal	full utilization of critical resource	balanced use of all resources
Reordering	bounded instruction balances	constant instruction balances

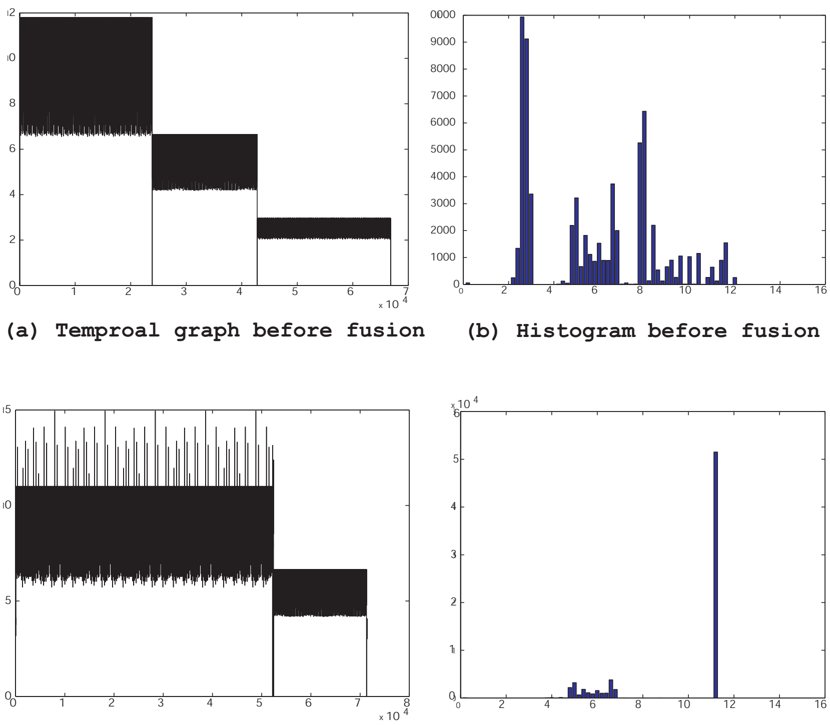
Table 2. Comparison of Program Reordering for Performance and for Energy

6 Evaluation

This section studies the effect of program reordering in a benchmark program, *Swim* from Spec95 suite. The program solves shallow water equations, a computation that is common in applications such as weather prediction. The main body of *Swim* consists of three loop nests enclosed in a time-step loop. Since different loops tend to have different instruction balances, we tried to combine them through loop fusion. We used a research compiler [7] that implements more aggressive loop fusion than current commercial compilers from Compaq, Sun, and SGI. It fused all three loop nests in *Swim*. Next we examine the effect of loop fusion on instruction balance.

We collected the execution trace and divided it into blocks of 500 instructions. For each block, we counted the number of different types of instructions and computed the balance. The trace collection and instruction enumeration were done on a Compaq Alpha 4000 Server using the ATOM tool [14]. To simplify the presentation, we will limit the discussion to only the ratio of floating-point to integer operations.

For this experiment, we ran the program in one iteration with an input size of 512x512. The original version has a total of 339 million instructions, out of which 166 million are floating-point and 43 million are integer operations. The upper two graphs of Figure 3 show the temporal graph and histogram of instruction balances. The temporal graph shows three segments, likely corresponding to the three loop nests. The ranges of instruction balances differ significantly in three segments: they are between 7 and 12, between 4 and 7, and between 2 and 3. The histogram shows the variation cumulatively. Instruction balances range from nearly 0 to 12 with high concentration points at 2.6 and 8.0 as well as a spread between 4.6 and 6.8. The largest single-point concentration covers no more than 15% of the program.



(a) Temproal graph before fusion (b) Histogram before fusion

(c) Temproal graph after fusion (d) Histogram after fusion

Fig. 3. Program balances of SPEC/Swim before and after loop fusion

Loop fusion altered the distribution of instruction balances, as shown by the lower two graphs in Figure 3. In the temporal graph, the number of segments has been reduced to two, one ranges between 6 and 12 and the other between 4 and 6. The histogram shows a dramatic change: over 70% of the program has an identical balance of 11.2, and all the rest are between 4.8 and 6.8. The histogram also shows that the seeming variation in the first segment in the temporal graph does not materialize in the histogram. Effectively, the fused version has a constant program demand in the first part of the computation. The number of instruction balances that are not 11.2 is too small to be seen on the histogram. Not shown in the figure, we also observed that loop fusion removed about 40% of integer operations because of the reduction in loop and data access overhead. The number of floating-point operations remained unchanged. Further study is needed to examine the full effect of program transformation, which is beyond the scope of this paper.

In summary, the study of *Spec/Swim* program has shown that program reordering can significantly alter program demand and re-balance the mix of instructions. For the ratio of floating-point to integer operations, program portion that has a constant balance is increased from under 15% to over 70%. The result of this experiment provides preliminary evidence that changing instruction balances is not only desirable but also feasible in a real application.

7 Related Work

A significant effort has been devoted to circuit or architectural improvement for power efficiency. Circuit-level features cannot be directly controlled by software, but architectural ones sometimes can. Our strategy depends on dynamic voltage and frequency scaling [2], which is being used by commercial processors such as Transmeta Crusoe and Intel XScale. Other architectural techniques such as clock gating [9] are also included in our model. Recently, Semeraro et al. studied fine-grained dynamic scaling, where funtional units within a processor may be scaled independently with each other [13]. The goal of our work is complementary and is to examine how can software better utilize these hardware features so that the whole system is optimized.

Software techniques have been studied for reducing energy usage [16, 15, 17]. 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 [17]. Better caching leads to power saving in the memory system. Vijaykrishnan et al. evaluated a set of cache optimizations by a compiler [18]. Most techniques save energy by reducing program demand. In contrast, we show the potential of demand reordering, which is targeted specifically at reconfigurable hardware. Demand reordering at the instruction level is the subject of two recent work. Both tried to spread out non-critical instructions and make program demand less concentrated. Greg et al. studied the potential benefit by rescheduling the execution trace [13]. Yang et al. modified software pipelining and measured its effect on the SPEC Integer benchmark suite [19]. Both reported double-digit percentage energy saving with no or little performance degradation.

In the past, power models at the CMOS level were studied for the processor core [2] and for memory hierarchy [5]. Instruction-level power consumption for fixed-configuration processors was measured for real machines [16, 12].

Architectural-level simulators have been developed [1, 18, 13]. Furthermore, researchers also studied better software feedback for reconfigurable hardware [10] and OS support for paging [11] and disk scheduling [8]. These techniques do not change the demand of software but improve the effectiveness of hardware adaptation. Techniques of program reordering like ours will benefit from power models and advanced system support.

Program optimization has been studied for many years for improving program performance. Here we review the ones that are related to instruction balance. Balance was introduced to model FPU throughput and load/store bandwidth [3]. Transformations such as unroll-and-jam are used to improve program balance [4]. To consider all levels of memory hierarchy, our earlier work extended the definition of balance from a ratio to a tuple [6]. In this paper, we further extend the definition to include all components of a computer including functional units within the CPU.

8 Conclusion and Future Work

This paper has presented a theoretical result to an important optimization problem regarding the best program organization on a machine of different components operating on a different set of frequencies. It has proved that a program with constant instruction balances consumes the least amount of energy, that balancing program instructions guarantees power saving without performance degradation, and that energy-based reordering is a different problem than performance-based reordering.

We are currently measuring instruction balance in programs and its exact role in energy consumption. We are also designing a *Smooth* compiler for improving instruction balances by building upon the global program and data transformations that we have developed previously [7].

Acknowledgment

The idea of this work was originated from a discussion with the PIs of the CAP project, Micheal Scott, Dave Albonesi, and Sandhya Dwarkadas. Bin Han provided an important hint that led to the proof of Lemma 1. We are also grateful to Xianghui Liu and members of the system group at the Computer Science Department of University of Rochester for their helpful discussions.

References

1. D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level analysis and optimization. In *Proceedings of the 27th International Symposium on Computer Architecture*, Vancouver, BC, 2000.
2. T. Burd and R. Brodersen. Processor design for portable systems. *Journal of LSI Signal Processing*, 13(2-3):203–222, 1996.

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. A. M. Despain and C. Su. Cache designs for energy efficiency. In *Proceedings of 28th Hawaii International Conference on System Science*, 1995.
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. Ding and K. Kennedy. Improving effective bandwidth through compiler enhancement of global cache reuse. In *Proceedings of International Parallel and Distributed Processing Symposium*, San Francisco, CA, 2001. http://www.ipdps.org.
8. F. Douglis, R. Caceres, B. Marsh, F. Kaashoek, K. Li, and J. Tauber. Storage alternatives for mobile computers. In *Proceedings of the first symposium on operating system design and implementation*, Monterey, CA, 1994.
9. S. Gary et al. PowerPC 603, a microprocessor for portable computers. In *IEEE Design and Test of Computers*, pages 14–23, 1994.
10. 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.
11. A. R. Lebeck, X. Fan, H. Zeng, and C. Ellis. Power aware page allocation. In *Proceedings of the 9th international conference on architectural support for programming languages and operating systems*, Cambridge, MA, 2000.
12. J. T. Russell and M. F. Jacome. Software power estimation and optimization for high performance, 32-bit embedded processors. In *Proceedings of International Conference on Computer Design*, Austin, Texas, 1998.
13. G. Semeraro, M. Grigorios, R. Balasubramonian, D. H. Albonesi, S. Dwarkadas, and M. L. Scott. Energy-efficient processor design using multiple clock domains with dynamic voltage and frequency scaling. Submitted for publication, 2001.
14. A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, Orlando Florida, June 1994.
15. C. Su, C. Tsui, and A. M. Despain. Low power architecture design and compilation techniques for high-performance processors. In *Proceedings of the IEEE COMPCON*, pages 489–498, 1994.
16. V. Tiwari, S. Maik, and A. Wolfe. Power analysis of embedded software: a first step towards software power minimization. *IEEE Transaction on VLSI Systems*, 1994.
17. V. Tiwari, S. Maik, A. Wolfe, and M. Lee. Instruction level power analysis and optimization of software. *Journal of VLSI Signal Processing*, 13(2):1–18, 1996.
18. N. Vijaykrishnan, M. Kandemir, M. J. Irwin, H. S. Kim, and W. Ye. Energy-driven integrated hardware-software optimizations using SimplePower. In *Proceedings of the 27th International Symposium on Computer Architecture*, Vancouver, BC, 2000.
19. H. Yang, G. R. Gao, and G. Cai. Maximizing pipelined functional unit usage for minimum power software pipelining. Technical Report CAPSL Technical Memo 41, University of Delaware, Newark, Delaware, September 2001.