# The Energy Impact of Aggressive Loop Fusion *

YongKang Zhu*, Grigorios Magklis‡, Michael L. Scott†, Chen Ding†, and David H. Albonesi*

* *Department of Electrical and Computer Engineering*
† *Department of Computer Science*
*University of Rochester*
*Rochester, New York, USA*
{*yozhu,albonesi*}*@ece.rochester.edu*
{*scott,cding*}*@cs.rochester.edu*

‡ *Intel Barcelona Research Center*
*Barcelona, Spain*
*grigoriosx.magklis@intel.com*

## Abstract

*Loop fusion combines corresponding iterations of different loops. It is traditionally used to decrease program run time, by reducing loop overhead and increasing data locality. In this paper, however, we consider its effect on* energy.

*By merging program phases, fusion tends to increase the uniformity, or* balance *of demand for system resources. On a conventional superscalar processor, increased balance tends to increase IPC, and thus dynamic power, so that fusion-induced improvements in program energy are slightly smaller than improvements in program run time. If IPC is held constant, however, by reducing frequency and voltage—particularly on a processor with multiple clock domains—then energy improvements may significantly exceed run time improvements.*

*We demonstrate the benefits of increased program balance under a theoretical model of processor energy consumption. We then evaluate the benefits of fusion empirically on synthetic and real-world benchmarks, using our existing loop-fusing compiler and a heavily modified version of the SimpleScalar/Wattch simulator. For the real-world benchmarks, we demonstrate energy savings ranging from 7–40%, with run times ranging from 1% slowdown to 17% speedup. In addition to validating our theoretical model, the simulation results allow us to "tease apart" the factors that contribute to fusion-induced time and energy savings.*

## 1. Introduction

With increasing concern over energy consumption and heat dissipation in densely-packed desktop and server systems, compiler optimizations that increase the energy efficiency of programs are becoming increasingly attractive. Among the most aggressive program transformations is loop fusion, which brings together multiple loops and interleaves their iterations. Two recent studies have shown significant performance benefits with loop fusion for a wide range of scientific programs [12, 24]. This paper studies the energy impact of loop fusion.

Loop fusion has two main effects on a program's demand for processor and memory resources. The first effect is to reduce demand, by reducing loop overhead and by increasing data reuse in registers and cache, which in turn reduces the number of memory operations and address calculations. The second effect is to *balance* demand, by combining loops with different instruction mixes, cache miss rates, branch misprediction rates, etc. Reduced demand naturally tends to save both time and energy. Increased balance also tends to save time and, to a lesser extent, energy, by increasing instruction-level parallelism (ILP): even with aggressive clock gating in inactive functional units, packing an operation into an unused slot in a superscalar schedule tends to save energy compared to extending program run time in order to execute the operation later. Beyond this obvious effect, however, we argue that increased balance has a special benefit for processors with *dynamic voltage scaling (DVS)*.

DVS allows CPU frequency and voltage to change dynamically at run time, in order to match demand. The Transmeta Crusoe TM5800 processor can scale its frequency from 800MHz down to 367MHz and its voltage from 1.3V down to 0.9V, thereby reducing power [13]. DVS is also employed on Intel XScale processors [10]. Because required voltage scales roughly linearly with frequency within typical operating ranges, and energy is proportional to the square of the voltage, a modest reduction in frequency can yield significant energy savings.

For rate-based (soft real-time) applications, and for applications that are I/O or memory bound, DVS allows the processor to slow down to match the speed of the real world or external bottleneck. Recently, researchers have proposed globally asynchronous, locally synchronous DVS processors in which the frequency and voltage of various processor components ("domains") can be changed independently at

*This work was supported in part by NSF under grants CCR-9988361, CCR-0204344, CCR-0219848, CCR-0238176, and EIA-0080124; by DoE under grant DE-FG02-02ER25525; by DARPA/ITO under AFRL contract F29601-00-K-0182; by an IBM Faculty Partnership Award; and by equipment grants from IBM, Intel and Compaq.

run time [18, 26, 27]. Such multiple clock domain (MCD) processors allow domains that are not on the processor's critical path to slow down to match the speed of an *internal* bottleneck.

On DVS processors, loop fusion can save energy even when it does not reduce demand or improve performance, as shown by the following example. The program in Figure 1(a) has two loops. Assume that memory is the performance bottleneck. Having the same memory demand, both loops take the same time $t$ for each iteration. With perfect frequency scaling, the two loops have CPU frequencies $10/t$ and $20/t$ respectively. Using a simplified energy model where power is a cubic function of frequency, the average CPU power is $((10/t)^3 + (20/t)^3)/2 = 4500/t^3$ units. Let us assume that the loops can be fused and, for illustration purposes, that loop fusion does not change the number of operations. In the fused loop, the ratio of CPU to memory operations is constant, as shown in Figure 1(b). The CPU frequency is now $30/2t$ or $15/t$, and the average power is $3375/t^3$, a 25% reduction, even though the reordered program executes the same operations in the same amount of time.

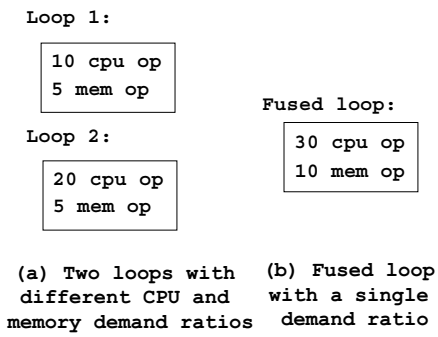


**Figure 1. Example of program reordering.**

While it ignores a host of issues (cache behavior, common subexpressions, ILP, etc.—all of which we address in section 4 by evaluating on real programs), this example suggests the opportunities for energy optimization available on DVS processors. While slowing down the processor to match external constraints (e.g. memory bandwidth or real-time deadlines) will almost always save energy, slowing the processor down *evenly* over a long period of time will save more energy than slowing it down a lot at some times and only a little at other times. Energy savings are maximized by good instruction balance—by even demand over time for each scalable processor component.

In the following section we present a mathematical model that captures the intuition of Figure 1, and describes the relationships among program balance, performance, and energy consumption in a perfect DVS processor. We prove that energy efficiency is maximized in a program with a constant ratio of demands for scalable processor components. In Section 3 we briefly review our loop-fusing compiler and multiple clock domain (MCD) processor (both of which are discussed in other papers), describe our application suite, and present the methodology we use to "tease apart" the various factors that lead to fusion-induced changes in program time and energy.

Our experimental evaluation appears in Section 4. We present run time and energy consumption, with and without loop fusion, on conventional and MCD processors. For the programs with fused loops, we also present a detailed breakdown of time and energy savings, attributing these savings to changes in the number of instructions executed, the number of cache misses, the number of mispredicted branches, the effectiveness of pipeline utilization, and the effectiveness of dynamic voltage scaling. We consider related work in more detail in Section 5, and conclude in Section 6.

## 2. Theory of program balance

We assume a processor/memory system consisting of $q$ components, each of which serves a different aspect of the program's resource needs (demand). We further assume that each component supports dynamic frequency and voltage scaling independent of other components. We divide the program's execution into a sequence of brief time intervals, each of duration $t$. We assume that during each interval $i$ the processor somehow chooses for each component $p$ an operating frequency $f_{pi}$ equal to $w_{pi}/t$, where $w_{pi}$ is the program's demand for $p$—the number of cycles of service on $p$ required in interval $i$. As an example, consider a pipelined integer unit $p$ that can finish one integer operation every cycle. If a program executes $100$ integer operations in a one-microsecond interval $i$, then $f_{pi} = 100/10^{-6} = 100$MHz.

Finally, we assume that the power consumption of a component $p$ running at frequency $f_{pi}$ during interval $i$ is $c(f_{pi})^k$, where $c$ is positive and $k$ is greater than 1. The energy used in $i$ is therefore $E_i = ct(f_{pi})^k$. Dynamic power is proportional to $V^2 \cdot f \cdot C$ where $V$ is the supply voltage, $f$ is the frequency of the clock, and $C$ is the effective switching capacitance [5]. We assume that voltage is scaled with frequency. Therefore, the constant $k$ can be as high as three. Different components may have a different constant $c$.

We do not consider any overhead incurred by dynamic scaling. Such overhead would not change our results: because demand is spread uniformly over time in the optimal case, overhead would be zero. We also choose not to consider dependences among components, or the fact that an individual instruction may require service from multiple components. We assume, rather, that components are completely independent, and that we can slide demand forward and backward in time, to redistribute it among intervals. Dependences prevent loop fusion and similar techniques from redistributing demand arbitrarily in real programs. Leaving

dependences out of the formal model allows us to evaluate program balance as a goal toward which practical techniques should aspire.

We define *program balance* in interval $i$ on a machine with $q$ components to be the vector $v_i = < w_{1i}, w_{2i}, \ldots, w_{qi} >$. We say that a program has *constant balance* if all the balance vectors are the same: i.e. $(\forall i, j)[v_i = v_j]$. A program has *bounded balance* if there is a critical component $p$ whose demand is always constant, and at least as high as the demand for every other component: i.e. $(\exists p, d)(\forall i, m)[(w_{pi} = d) \wedge (w_{mi} \leq d)]$. The critical component $p$ is the one with the largest ratio of its total demand to its highest frequency. Constant balance implies a constant utilization of all components. Bounded balance guarantees only the full utilization of the critical component. The utilization of other components may vary. The difference will become significant when we compare the use of reordering for performance and for energy.

**The basic theorem of program balance.** Assuming that time is held constant, we argue that a program execution consumes minimal energy if it has constant balance. Put another way, a program with varied balance can be made more energy efficient by smoothing out the variations. Since we are assuming that components are independent, we simply need to prove this statement for a single component. We therefore drop the $p$ subscripts on the variables in the following discussion.

An execution comprising $M$ intervals clearly uses time $T = Mt$. It has a total demand $W = \sum_{i=1}^{M} w_i = \sum_{i=1}^{M} f_i t = t \sum_{i=1}^{M} f_i$, and consumes energy $E_{original} = \sum_{i=1}^{M} c(f_i)^k t = ct \sum_{i=1}^{M} (f_i)^k$. The balanced execution has the same running time $T$ and total demand $W$. It uses a constant frequency $f = W/T$, however, and consumes energy $E_{balanced} = cTf^k$.

**Theorem 1** *The following inequality holds.*

$$E_{original} = ct \sum_{i=1}^{M} (f_i)^k \geq cTf^k = E_{balanced}$$

*where c, t, and $f_i$ are non-negative real numbers, and $k$ and $M$ are greater than 1.*

*Proof.* According to a special form of Jensen's inequality (Jensen's theorem) [19],

$$\frac{\sum_{i=1}^{M} h(f_i)}{M} \geq h(\sum_{i=1}^{M} \frac{f_i}{M})$$

where $h(f_i)$ is a convex function for the range of $f_i$. Let $h(f_i) = f_i^k$, which is a convex function when $f_i \geq 0$ and $k \geq 1$. It follows that $ct \sum_{i=1}^{M} (f_i)^k \geq ctMf^k = cTf^k$. □

The theorem assumes that the operating frequency of a machine can be any real number. If a machine can choose only from a set of predefined integral operating frequencies, Jensen's theorem is not directly applicable. Li and Ding [22] used another calculus method and proved that in the case of discrete operating frequencies, the optimal program balance is achieved by a strategy that alternates between the two closest valid frequencies above and below $W/T$, where $T$ is the running time and $W$ is the total demand.

**Reordering for energy vs. performance.** From a performance perspective, the goal of reordering is to fully utilize the critical resource, thereby requiring bounded balance. For example, if memory is the critical resource, we can maximize performance by keeping the memory busy at all times. The utilization of the CPU does not matter as long as it does not delay the demand to memory. For energy, however, we want not only full utilization of the critical resource but also a constant utilization of all other resources. According to the theorem of program balance, a program with varied CPU utilization consumes more energy than an equivalent program with constant CPU utilization. This is the reason for the energy improvement in the example given in the introduction: execution time cannot be reduced but energy can.

## 3. Experimental methodology

### 3.1. Loop fusion

We use the source-to-source locality-based fusion algorithm devised by Ding and Kennedy to improve the effectiveness of caching [12]. A similar approach was used in the Intel Itanium compiler [24]. We chose this algorithm because it is familiar to us, its implementation is available, and it outperforms the commercial compiler for our target platform (the Compaq Alpha) on our application suite.

Unlike most previous work, locality-based fusion is able to fuse loops of differing shape, including single statements (loops with zero dimension), loops with a different number of dimensions, and imperfectly nested loops. It uses a heuristic called *sequential greedy fusion*: working from the beginning of the program toward the end, the algorithm fuses each statement or loop into the earliest possible data-sharing statement or loop. For multi-level loops, the algorithm first decides the order of loop levels and then applies single-level fusion from the inside out. This heuristic serves to minimize the number of fused loops at outer levels. Unlike Ding and Kennedy, we fuse loops for the sake of program balance even when doing so does not improve data reuse—e.g. even when the loops share no data. Function in-lining is used where possible to fuse loops across function boundaries.

Aggressive fusion may cause register spilling and increased cache interference. The register spilling problem can be solved by constrained fusion for innermost loops [11, 30]; the problem of cache interference can be alleviated by data regrouping, which places simultaneously accessed data into the same cache block to avoid cache conflicts [12].

Locality-based fusion follows the early work of vectorizing compilers [3], which applies maximal loop distribution before fusion. It subsumes a transformation known as loop fission, which splits a loop into smaller pieces to improve register allocation.

### 3.2. MCD architecture and control

We use the multiple clock domain (MCD) processor described by Semeraro et al. [27]. MCD divides the chip into four domains: the **fetch** domain (front end), which fetches instructions from the L1 I-cache, predicts branches, and then dispatches instructions to the different issue queues; the **integer** domain, which comprises the issue queue, register file, and functional units for integer instructions; the **floating-point** domain, which comprises the same components for floating-point instructions; and the **memory** domain, which comprises the load-store queue, L1 D-cache, and unified L2 cache. Each domain has its own clock and voltage generators, and can tune its frequency and voltage independent of the other domains. Architectural queues serve as the interfaces between domains, and are augmented with synchronization circuitry to ensure that signals on different time bases transfer correctly. This synchronization circuitry imposes a baseline performance penalty of approximately 1.3% on average.

Previous papers describe three control mechanisms to choose when, and to what values, to change domain frequencies and voltages. The *off-line algorithm* [27] post-processes an application trace to find, for each interval, the configuration parameters that would have minimized energy, subject to a user-selected acceptable slowdown threshold. Though impractical, it provides a target against which to compare more realistic alternatives. The *on-line algorithm* [26] makes decisions by monitoring the utilization of the issue queues, which also serve to communicate among domains. The *profile-based algorithm* [23] uses statistics gathered from one or more profiling runs to identify functions or loop nests that should execute at particular frequencies and voltages. The results in this paper were obtained with the off-line algorithm, with a slowdown target of 2%. The profile-based algorithm achieves equally good energy savings, but would have made it significantly more cumbersome to conduct our experiments.

Our simulation results were obtained with a heavily modified version of the SimpleScalar/Wattch toolkit [4, 6]. Details can be found in other papers [23, 26, 27]. Architectural parameters were chosen, to the extent possible, to match those of the Alpha 21264 processor. Main memory was always run at full speed, and its energy was not included in our results.

### 3.3. Application suite

Most existing compilers are unable to fuse loops of different shape. As a result, they tend to obtain little or no benefit from loop fusion in real applications [9]. Our compiler does a bit better, successfully fusing large-scale loops in five of fifty benchmarks from standard benchmark suites: Livermore Loops *LK14* and *LK18*, *Swim* and *Tomcatv* from SPEC95, and *SP* from the NAS suite. For the other 45 benchmarks in these suites our compiler is a no-op. We also report results for *ADI*, a kernel for alternating-direction integration, which has been widely used as a benchmark in the locality optimization literature. Loop fusion significantly improves the performance of all these applications.

Recent work by Ng et al. (to which we do not have access) has extended locality-based fusion to cover additional programs, via procedure inlining and array contraction [24]. The authors report an average speedup of 12% on the 14 applications of the SPECfp2K suite, running on an Intel Itanium processor. With continued improvements in the state of the art, it seems reasonable to expect that fusion will become a widely used technique.

Our compiler fuses most programs into a single loop nest. The exceptions are *SP*, in which over 100 original loop nests are fused into a dozen new loops, and *Tomcatv*, in which data dependences allow only the outermost loops to be fused. As noted above, aggressive fusion may increase register spilling and cache interference. In our test suite spilling increases only in *SP*. The problem can be solved by constrained fusion for innermost loops (equivalent to loop distribution after fusion) [11, 30], but we did not implement the transformation for this work; the modest potential benefits in *SP* did not seem to warrant the effort. Increased cache conflicts also appeared in only one of our applications, but here the impact was substantial: *Swim* was the only application that ran slower after fusion. We used Ding and Kennedy's data regrouping algorithm [12] to eliminate the new cache conflicts, at which point the fused version of *Swim* ran 18% faster than the original.

We compiled all benchmark programs (with or without prior source-to-source loop fusion) using the Digital f77 compiler with the `-O5` flag. The machine compiler performed loop transformations and software pipelining. We chose input sizes for our experiments so that the total number of instructions simulated for each benchmark was around 100 million.

### 3.4. Time and energy breakdown

Given differences in run time and energy consumption between the original and fused versions of an application, we would like to determine the extent to which these differences are the simple result of changes in instruction demand (number of instructions executed) and, conversely, the extent to which they stem from differences in the effectiveness of such architectural features as branch prediction, caching, pipeline utilization, and voltage scaling.

Architectural effects, of course, may not be entirely independent. Memory accesses on mispredicted paths, for example, may change the cache hit rate, while timing changes due to hits and misses in the cache may change the distance that the processor travels down mispredicted paths. Still, we can gain a sense of the relative significance of different factors by considering them in order, and can quantify their contributions to run time and energy by running our applications on various idealized machines.

Suppose the original program runs in time $T_c^o$ and consumes energy $E_c^o$, where the superscript "o" stands for "original" and the subscript "c" stands for a conventional simulated machine—not idealized in any way. Similarly, suppose the fused program runs in time $T_c^f$ and consumes energy $E_c^f$. We would like to attribute the difference in run time $\Delta T_c = T_c^o - T_c^f$ and energy $\Delta E_c = E_c^o - E_c^f$ to differences in (a) the effectiveness of branch prediction, (b) the effectiveness of caching, (c) the effectiveness of pipeline utilization, (d) the level of instruction demand, and (e) the effectiveness of DVS.

**Synchronous case.** Consider first the issues of branch prediction and caching. We can simulate the original and fused programs on a machine with perfect branch prediction and, optionally, perfect caching. We use the subscript $c, pb$ to indicate execution on a conventional (synchronous) machine with perfect branch prediction, $c, pc$ to indicate execution on a conventional machine with perfect caching (all data accesses hit in the L1 D-cache), and $c, pb, pc$ to indicate execution on a conventional machine with both. The differences $T_c^o - T_{c,pb}^o$ and $E_c^o - E_{c,pb}^o$ represent the time and energy spent on mispredicted paths in the original program. Similarly $T_c^f - T_{c,pb}^f$ and $E_c^f - E_{c,pb}^f$ represent the time and energy spent on mispredicted paths in the fused program. We thus define

$$\begin{aligned}\Delta T_{prediction} &= (T_c^o - T_{c,pb}^o) - (T_c^f - T_{c,pb}^f)\\ \Delta E_{prediction} &= (E_c^o - E_{c,pb}^o) - (E_c^f - E_{c,pb}^f)\end{aligned}$$

These are the *extra* time and energy spent in the original program executing useless instructions. In a similar vein, $T_{c\text{-}pb}^o - T_{c\text{-}pb,pc}^o$ and $E_{c\text{-}pb}^o - E_{c\text{-}pb,pc}^o$ are the time and energy spent servicing cache misses in the original program, in the absence of mispredictions, and $T_{c\text{-}pb}^f - T_{c\text{-}pb,pc}^f$ and $E_{c\text{-}pb}^f - E_{c\text{-}pb,pc}^f$ are the analogous quantities for the fused program. We thus define

$$\begin{aligned}\Delta T_{caching} &= (T_{c,pb}^o - T_{c,pb,pc}^o) - (T_{c,pb}^f - T_{c,pb,pc}^f)\\ \Delta E_{caching} &= (E_{c,pb}^o - E_{c,pb,pc}^o) - (E_{c,pb}^f - E_{c,pb,pc}^f)\end{aligned}$$

These are the *extra* time and energy spent in the original program servicing cache misses in the absence of mispredictions. They include, by design, misses due both to poorer register reuse (i.e., extra loads) and to poorer hardware cache locality.

After factoring out mispredictions and misses, we attribute any remaining CPI and EPI differences between the original and fused programs to the effectiveness of pipeline packing. This gives us

$$\begin{aligned}\Delta T_{pipeline} &= (CPI_{c,pb,pc}^o - CPI_{c,pb,pc}^f) \times N^f\\ \Delta E_{pipeline} &= (EPI_{c,pb,pc}^o - EPI_{c,pb,pc}^f) \times N^f\end{aligned}$$

where $N^f$ is the number of instructions committed by the fused program.[1] These definitions are approximate, in the sense that they do not reflect the fact that instructions present in the original program but not in the fused program may be disproportionately faster or slower than the average across the whole program.

Remaining differences we attribute to instruction demand:

$$\begin{aligned}\Delta T_{inst\_dem} &= \Delta T_c - \Delta T_{prediction} - \Delta T_{caching} - \Delta T_{pipeline}\\ \Delta E_{inst\_dem} &= \Delta E_c - \Delta E_{prediction} - \Delta E_{caching} - \Delta E_{pipeline}\end{aligned}$$

**MCD case.** Now consider the case of an MCD processor with dynamic frequency and voltage scaling. Because DVS is an energy saving technique, not a performance enhancing technique, and because the off-line control algorithm chooses frequencies with a deliberate eye toward bounding execution slowdown, it makes no sense to attribute differences in run time to differences in the "effectiveness" of DVS. We therefore focus here on energy alone.

By analogy to the synchronous case, let $E_m^o$ and $E_m^f$ be the energy consumed by the original and fused programs running on an MCD processor with frequencies and voltages chosen by the off-line algorithm. For the original program, DVS reduces energy by the factor $r = E_m^o/E_c^o$. If

[1] We can (and, in our reported results, do) obtain a better estimate of $\Delta E_{pipeline}$ by taking into account instruction mix and the energy consumptions in separate hardware units. Given the instruction mix and that our simulator reports energy numbers separately for the following units: the front end (used by all instructions), the integer unit (used by all non-floating point instructions), the branch predictor, the floating point unit, and the memory unit, we can calculate separate EPI for each unit for the original and fused programs, determine the $\Delta E_{pipeline}$ contribution from each unit by the equation shown in the text, and then add up all these contributions to obtain the total $\Delta E_{pipeline}$. This refinement, of course, is not possible for time.

the various other executions measured in the synchronous case made equally good use of DVS, we would expect their energies to scale down by this same factor $r$. We therefore define

$$\begin{aligned}\Delta E_m &= E_m^o - E_m^f \\ &= r \times (\Delta E_{inst\_dem} + \Delta E_{caching} \\ &\qquad + \Delta E_{prediction} + \Delta E_{pipeline}) + \Delta E_{dvs}\end{aligned}$$

where $\Delta E_{inst\_dem}$, $\Delta E_{caching}$, $\Delta E_{prediction}$, and $\Delta E_{pipeline}$ are all calculated as in the synchronous case. Equivalently:

$$\Delta E_{dvs} = \Delta E_m - r\Delta E_c = (E_m^o - E_m^f) - r\,(E_c^o - E_c^f)$$

Under this definition, $\Delta E_{dvs}$ is likely to be negative, because loop fusion tends to reduce cache misses and CPI, thereby reducing opportunities to save energy by lowering frequency and voltage. The balance theorem of section 2, however, suggests that fusion should *increase* the effectiveness of DVS *when overall time is kept constant*. To evaluate this hypothesis, we will in Section 4.3 consider executions in which we permit the off-line algorithm to slow down the fused program so that it has the same run time it would have had without fusion-induced improvements in pipeline utilization and, optionally, caching.

## 4. Evaluation

In this section we first consider a contrived test program in which improved instruction balance allows an MCD processor to save energy even when it does not save time. We then consider the impact of loop fusion on execution time and energy for the benchmark applications described in Section 3.3. For each of these benchmarks, we use the methodology of Section 3.4 to attribute time and energy savings to changes in instruction demand and in the effectiveness of caching, branch prediction, pipeline utilization, and (for energy) DVS.

### 4.1. Saving energy without saving time

Figure 2 shows a simple kernel program with two loops. To balance this program, we can move the statement labeled *S* from the second loop to the end of the first, thus giving both loops the same mix of integer and floating-point operations. The program is written in an assembly-like style and compiled with minimal optimization (`-O1`) to make the object code as straightforward and predictable as possible.

Since the two loops operate on different arrays, moving statement `S` from the second loop to the first has little impact on instruction count or cache performance. Indeed, as shown in the *Test* column of Table 2, the original and modified versions of the program have nearly identical counts for all types of instructions, and the same number of cache misses. Execution time on the MCD machine is also essentially the same. Energy consumption, however, is reduced by almost 6%. Figure 3 shows the frequencies selected over time by the off-line algorithm for the original and balanced versions of *Test* on the MCD processor. In the integer, floating-point, and memory domains, frequency is much more stable and smooth after balancing. Frequency in the fetch domain remains essentially constant.

```
/* N, U, V, and W are constants */
unsigned long long P[N], Q[N];
double c, d, e, f;
unsigned long long *base_p, *org_p, *end_p;
unsigned long long *base_q, *org_q, *end_q;

base_p = org_p = (unsigned long long *)P + 2;
base_q = org_q = (unsigned long long *)Q + 2;

end_p = (unsigned long long *)P + N;
end_q = (unsigned long long *)Q + N;

for( i = 0; i < 3; i++ )
{
      /* the first loop */
L1:   *base_p = *(base_p-1) * Y - *(base_p-2);
      base_p++;
      c = c + W;
      if( base_p < end_p )
         goto L1;

      /* the second loop */
L2:   *base_q = *(base_q-1) * Z - *(base_q-2);
      base_q++;
      d = d + W;
      e = e + U;
S:    f = f + V;          /* to be moved
            to the end of the first loop */
      if( base_q < end_q )
         goto L2;

      base_p = org_p;
      base_q = org_q;
}
```

**Figure 2. A contrived *Test* program. As written, the two loops have the same number of multiplications and memory operations per iteration, but different numbers of additions.**

| *Test* | baseline machine | MCD |
|---|---|---|
| fetch domain | 0.00% | −0.73% |
| integer domain | 0.13% | 2.84% |
| floating point domain | 0.16% | 18.78% |
| memory domain | 0.02% | 3.77% |
| **total energy reduction** | **0.08%** | **5.74%** |

**Table 1. Energy benefit of balancing in *Test*, on the baseline and MCD processors.**

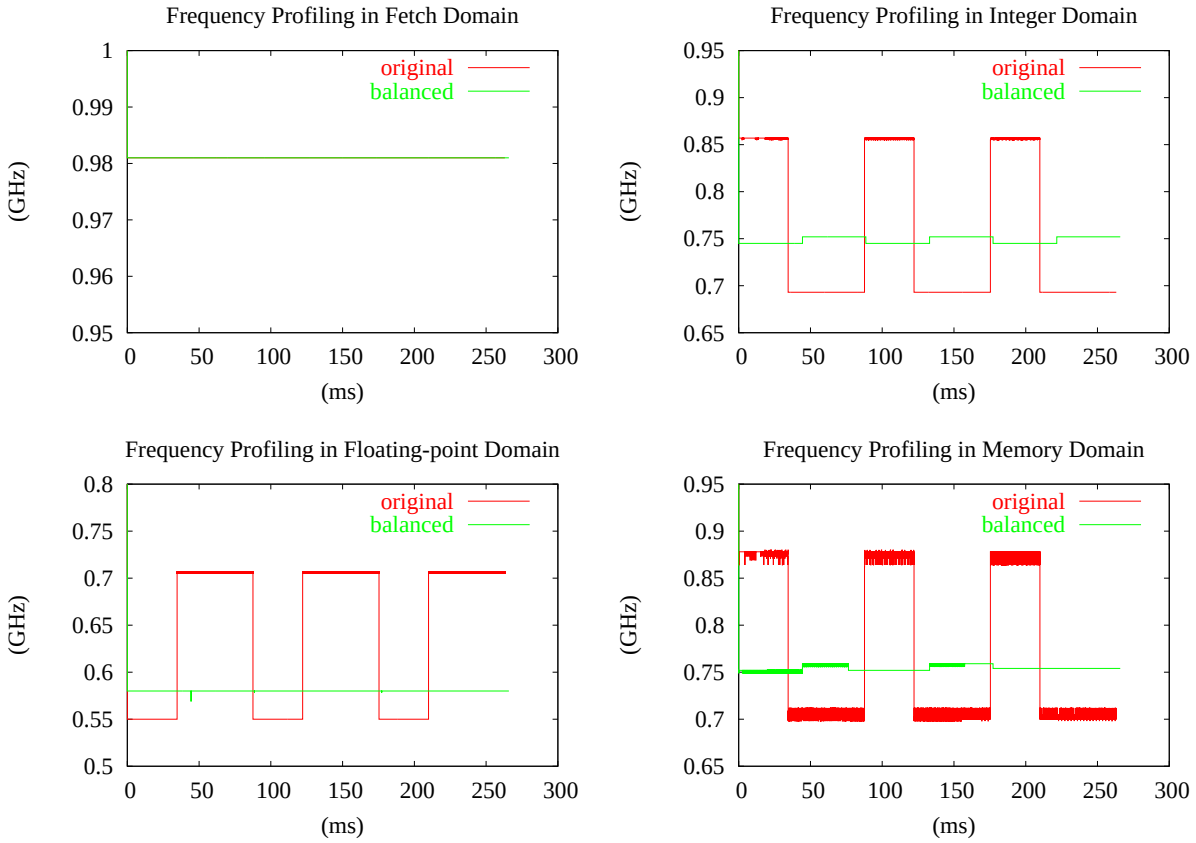


**Figure 3. Frequency curves for *Test*. In the fetch domain, the two curves lie on top of each other.**

Table 1 shows the impact of moving statement `S` on the energy consumption of the baseline (globally synchronous) and MCD processors. Without DVS, the overall change in energy is below one tenth of a percent, and no individual domain sees a change of more than 0.16%. On the MCD processor, however, where DVS allows us to adjust frequency and voltage to match the demands of the program, the more balanced version of the program admits overall energy savings of over 5%, mostly by allowing the floating-point unit to run more slowly in loop `L1`. This example demonstrates clearly that with DVS, program balancing can save energy even when it doesn't change workload or execution time.

## 4.2. Program balance in realistic benchmarks

Table 2 shows statistics both for *Test* and for the six real benchmarks, on the MCD processor. Reductions in energy due to loop fusion range from −0.4% to 28%, a bit lower than the reductions in run time, which range from 4.7% to 36.9%. The reductions in run time can be attributed to reductions in the total number of instructions (four of the six real benchmarks) and reductions in cycles per instruction (CPI—five of the six real benchmarks).

Energy per cycle (average power) increases by amounts ranging from 5.3% to 16.5% in five of the six real benchmarks. The fused program incurs higher average power when it runs faster due to the better loop-level parallelism after loop fusion. *LK14* shows a 0.3% decrease in average power, consistent with its small increase in CPI. *Test* shows a 6.7% decrease in power, due to better balance over constant time. Energy per instruction (EPI) shows somewhat smaller changes, increasing slightly in *LK14* and *SP*, and decreasing by modest amounts in the other programs.

Interestingly, while *Tomcatv* executes 1.6% *more* dynamic instructions after fusion, and suffers both a larger number of memory operations and a larger number of misses, it still runs 4.7% faster, mainly due, we believe, to the elimination of misses on the critical path. These eliminated misses are reflected in a dramatic reduction in the standard deviation of the frequency chosen by the off-line algorithm for the floating-point domain. As noted in Section 3.3, *Tomcatv* is the only benchmark for which our compiler was able to fuse outer but not inner loops.

In all benchmarks other than *Tomcatv*, fusion leads to significant improvements in both L1D and L2 cache miss rates. It also tends to reduce the fraction of integer and/or

| | Test | ADI | LK14 | LK18 | SP | Swim | Tomcatv |
|---|---|---|---|---|---|---|---|
| exe time[1] | −1.0% | 36.9% | 27.8% | 7.5% | 26.6% | 17.1% | 4.7% |
| energy[1] | 5.7% | 26.5% | 28.0% | 0.7% | 18.1% | 11.1% | −0.4% |
| total inst[1] | 0.0% | 20.4% | 29.8% | −0.2% | 18.6% | 6.0% | −1.6% |
| CPI[2] | 2.05(−1%) | 1.11(21%) | 0.80(−3%) | 1.26(8%) | 0.50(10%) | 0.89(12%) | 1.08(6%) |
| EPI[1] | 5.7% | 7.7% | −2.7% | 0.9% | −0.3% | 5.5% | 1.1% |
| avg power[1] | 6.7% | −16.5% | 0.3% | −7.4% | −11.5% | −7.3% | −5.3% |
| inst mix org[3] | 50/8/37/4 | 7/34/57/2 | 27/17/55/1 | 7/46/46/1 | 15/32/50/2 | 12/40/48/1 | 8/43/46/3 |
| inst mix fused[3] | 50/8/37/4 | 8/42/50/1 | 18/25/56/1 | 10/46/43/0 | 9/40/51/1 | 2/42/56/0 | 7/42/49/3 |
| L1D miss[4] | 3% (0%) | 9% (32%) | 5% (45%) | 6% (25%) | 7% (22%) | 4% (15%) | 7% (−1%) |
| L2 miss[4] | 50% (0%) | 43% (58%) | 59% (45%) | 57% (32%) | 1% (65%) | 63% (29%) | 80% (−6%) |
| num reconfi g[2] | 2111 (87%) | 10814 (48%) | 3937 (71%) | 18379 (19%) | 4727 (37%) | 8142 (33%) | 23137 (-6%) |
| freq-stdev[5] | 5.83/0.01 | 10.34/25.53 | 4.88/4.23 | 36.39/0.23 | 9.96/3.91 | 31.25/20.68 | 19.28/1.89 |

1 : percentage reduction after fusion/balancing

2 : value after fusion/balancing (and percentage reduction)

3 : percentage of integer/fbating-point/memory-reference/branch instructions

4 : local miss rate after fusion/balancing (and percentage reduction in total number of misses)

5 : standard deviation of fbating point frequency (MHz) before/after fusion/balancing

**Table 2. Simulation results on MCD processor (with off-line frequency selection) before and after loop fusion (six real benchmarks) or balancing (*Test*).**

memory instructions, leading to significant increases in *ADI*, *LK14*, and *SP* in the fraction of the remaining instructions executed by the floating-point unit. In all benchmarks other than *ADI*, fusion also leads to significant reductions in the standard deviation of the frequency chosen by the off-line algorithm for the floating-point domain. We believe the difference in *ADI* stems from a mismatch between the size of program loops and the 10,000 cycle window size used by the off-line control algorithm [27]. In separate experiments (not reported here), the on-line control algorithm of Semeraro et al. [26] enabled fusion in ADI to reduce the standard deviation of the floating-point domain by a factor of 2.4. In all benchmarks other than *Tomcatv*, fusion leads to a significant reduction in the total number of frequency changes (reconfigurations) requested by the off-line algorithm.

Figure 4 illustrates the synergy between loop fusion and DVS. Without MCD scaling, loop fusion reduces program energy by 3–32%, due entirely to reductions in run time of 6–36%. Without loop fusion, MCD scaling reduces program energy by 12–23%, but increases program run time by 2–11%. Together, MCD scaling and loop fusion achieve energy savings of 12–43%, while simultaneously improving run time by 2–34%. While it is certainly true that reductions in frequency and voltage will save energy in almost any program, the "bang for the buck" tends to be higher in a fused program than it was in the original, because (as shown in Table 2) the fused program's power is higher. Please note that, the bars in Figure 4 are not directly comparable to the top two rows of Table 2: improvements in Figure 4 are measured with respect to the globally synchronous processor, while those in Table 2 are measured with respect to the MCD processor.

### 4.3. Breakdown of time and energy improvements

Using the methodology of Section 3.4, we can estimate the extent to which fusion-induced reductions in run time and energy can be attributed to reductions in instruction count and to improvements in the effectiveness of caching, branch prediction, pipeline utilization, and DVS. In some cases, of course, the "reductions" or "improvements" may be negative. In *Tomcatv*, for example, the fused program executes more instructions than the original.

Figure 5 illustrates the breakdown of run time and energy improvements on a synchronous (non-DVS) processor. Total improvements are the same as the leftmost bars in each group in Figure 4. Since all programs have regular loop structures, the branches are easy to predict before and after loop fusion. We found $\Delta T_{prediction}$ and $\Delta E_{prediction}$ to be essentially zero in all cases, so we have left them out of the graphs. While reduction in instruction demand accounts for about 50% of the run time savings in *LK14* and *SP*, it has almost no effect in *LK18* and is negative in *Tomcatv*. This is consistent with the fact that *LK18* and *Tomcatv* both have more instructions after fusion. Pipeline effectiveness is the dominant factor in *ADI* and *LK18*, and is the least important one in *LK14* and *SP*. *Swim* is the only real benchmark that does not benefit from pipeline effectiveness, which is consistent with the fact that its IPC drops after fusion in the execution with perfect caches and perfect branch prediction. Better caching brings benefit to all the six real benchmarks and particularly, it accounts for almost all the run time savings in *Swim*. The *Test* program, as expected, sees no change in run time due to fusion. Energy numbers are similar though not identical, and the relative importance

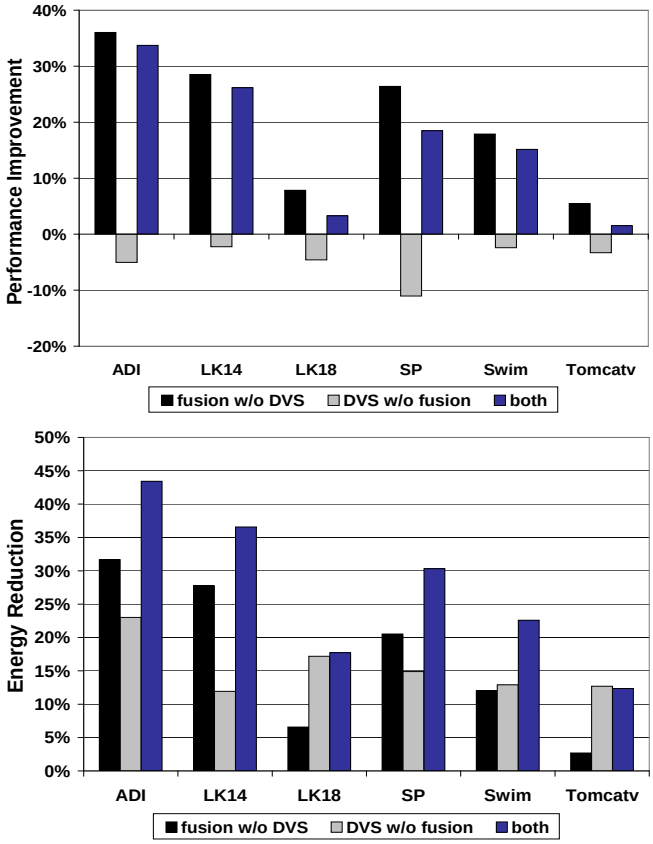


**Figure 4. Effects on energy and performance from fusion alone, from dynamic voltage scaling alone, and from both together. The baseline for comparison is the globally synchronous processor, without inter-domain synchronization penalties.**

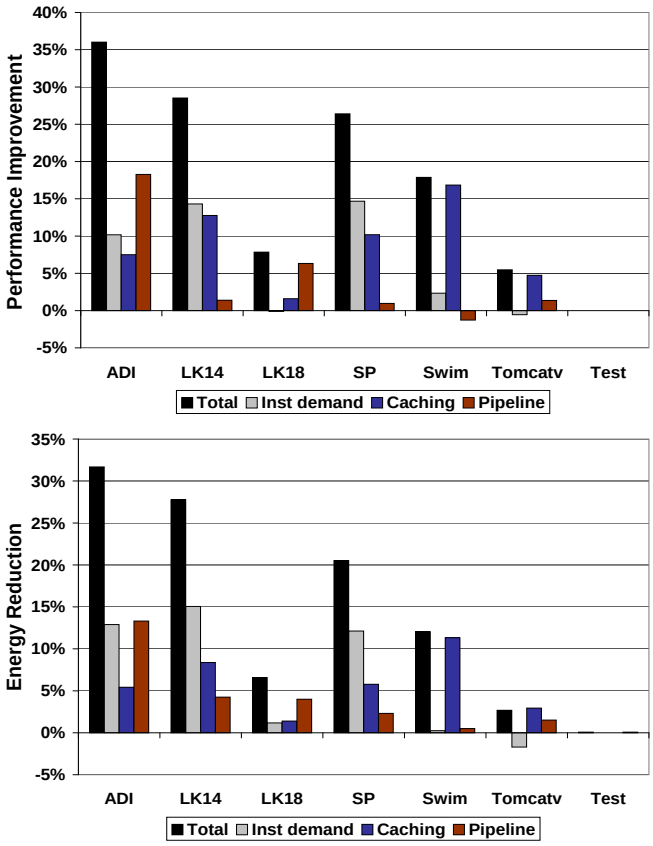


**Figure 5. Breakdown of reduction in run time (top) and energy (bottom) between original and fused/balanced programs on synchronous processor.**

of caching and pipeline effectiveness changes for several applications.

Figure 6 illustrates the breakdown of energy improvements on the MCD processor, where we have added a bar to each application for $\Delta E_{dvs}$. In five of the six real benchmarks, fusion *reduces* the effectiveness of DVS by eliminating opportunities to slow down clock domains that are off the critical path. As noted at the end of Section 3.4, however, we have the opportunity on a machine with DVS to save energy by slowing execution, and loop fusion enhances this opportunity by eliminating many of the cache misses of the original program.

Figure 7 presents values of $\Delta E_{dvs}$ for three different execution models. The first bar in each group is the same as in Figure 6: it represents the fused program running on an MCD processor at frequencies and voltages chosen by the off-line algorithm with a target slowdown of 2%. The second bar represents the energy savings due to MCD scaling when we slow execution just enough to "use up" the fusion-induced savings in run time due to better caching. The last bar shows the corresponding savings when we slow execution enough to use up the benefits due both to better pipeline packing *and* to better caching. More specifically, for "dilated" and "double dilated" executions, we let

$$T_d^f = T_m^f + \Delta T_{caching}$$
$$T_{dd}^f = T_d^f + \Delta T_{pipeline}$$

The definition of $T_d^f$ is based on the assumption that the time spent waiting for cache misses is essentially unaffected by MCD scaling. If we were to duplicate all of Figure 6 for each of the time-dilated cases, the values of $\Delta E_{inst\_dem}$, $\Delta E_{caching}$, $\Delta E_{prediction}$, and $\Delta E_{pipeline}$ would remain unchanged; total energy savings would increase by the same amount as $\Delta E_{dvs}$. For the six real benchmarks, $\Delta E_{dvs}$ ranges from $-2$–35% when we scale down processor frequency to recoup the time saved by better caching. It ranges from 7–40% when we also recoup the time saved by better pipeline packing. Even in this extreme case, the fused versions of *LK18* and *Tomcatv* run within 1% of the time of the original program, and *ADI*, *LK14*, *SP* and *Swim* run from 3–17% faster. For *Swim*, there is less $\Delta E_{dvs}$ savings in its double-dilated execution than the dilated one. This is be-

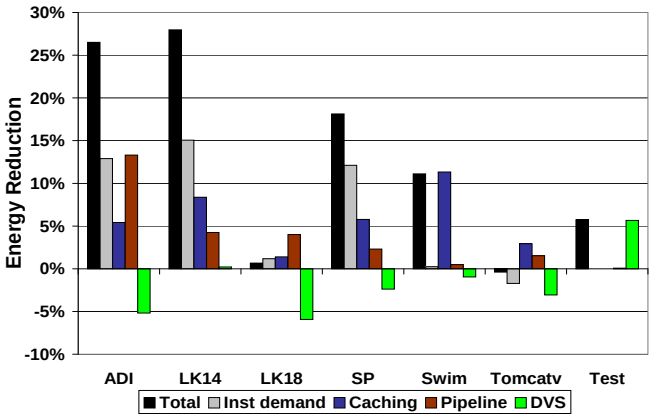


**Figure 6. Breakdown of reduction in energy between original and fused/balanced programs with MCD frequency/voltage scaling. DVS is *less* effective in the fused version of five out of six real benchmarks.**

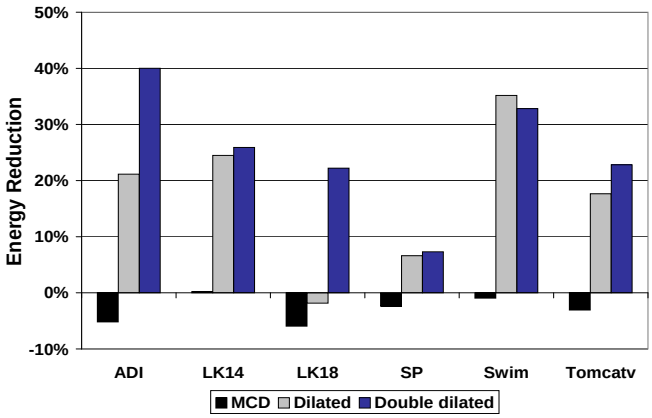


**Figure 7. Energy savings in loop-fused application attributable to increased effectiveness of DVS, with standard 2% MCD target slowdown (left), and with additional slowdown sufficient to consume $\Delta T_{caching}$ (middle) and, optionally, $\Delta T_{pipeline}$ as well (right).**

cause its run time savings due to fusion-induced "better" pipeline packing is negative, as shown in Figure 5.

## 5. Related work

**Loop fusion.** Many researchers have studied loop fusion. Early work includes that of Wolfe [31] and of Allen and Kennedy [2]. Combining loop fusion with distribution was originally discussed by Allen et al [1]. Global loop fusion was formulated as a graph problem by Allen et al. [1] for parallelization, by Gao et al. [14] for register reuse, and by Kennedy and McKinley [21] for locality and parallelization. Loop fusion has been extended and evaluated in many other studies that we do not have space to enumerate. Ding and Kennedy used reuse-based fusion to fuse loops of different control structures [12]. Ng et al. enhanced reuse-based loop fusion with locality-conscious procedural inlining and array contraction and implemented it in the Intel Itanium compiler [24]. Pingali et al. generalized the idea of loop fusion to computation regrouping and applied it manually to a set of C programs commonly used in information retrieval, computer animation, hardware verification and scientific simulation [25]. Strout et al. developed sparse tiling, which effectively fuses loops with indirect array access [29]. Where these fusion studies were aimed at improving performance on conventional machines, our work is aimed at saving energy on DVS processors. As a result, we fuse loops even when their data are disjoint.

**Program balancing.** Callahan et al. defined the concepts of program and machine balance [7]. For single loop nests, Carr and Kennedy used program transformations such as scalar replacement and unroll-and-jam to change the program balance to match the balance of resources available on the machine [8]. So et al. used balance information to restrict the search to appropriate matches between hardware design and loop optimization, consequently making hardware and software co-design much faster [28]. Similar to the work of So et al., we target adaptive hardware that has no fixed machine balance. But unlike the work of So et al. and all other previous work, we do not require separate loop nests to have a particular balance; we only want them to have the *same* balance.

**Dynamic voltage and frequency scaling.** Dynamic voltage and frequency scaling was studied by Burd and Brodersen [5]. Semeraro et al. designed a multiple clock domain (MCD) system to support fine-grained dynamic voltage scaling within a processor [27]. A similar design was proposed by Iyer and Marculescu [18]. Various schemes have been developed to control adaptive processors, including compiler assisted methods by Hsu et al. [16], an on-line method by Semeraro et al. [26], and methods using both compiler and profiling analysis by Magklis et al. [23] and by Hsu and Kremer [15].

Yao et al. [34] and Ishihara and Yasuura [17] studied the optimal schedule for DVS processors in the context of energy-efficient task scheduling. Both showed that it is most energy efficient to use the lowest frequency that allows an execution to finish before a given deadline. Ishihara also showed that when only discrete frequencies are allowed, the best schedule is to alternate between at most two frequencies.

On a multi-domain DVS processor, optimal scheduling is not possible if a program has varying demands for different domains at different times. The theorem in our paper shows that in this case, one needs to fully balance a pro-

gram to minimize its energy consumption. Li and Ding first proved this theorem for DVS systems with continuous and discrete operating frequencies [22]. This paper generalizes the basic theorem for any power function $P \propto f^k$, where $f$ is frequency and $k > 1$. Ishihara and Yasuura assumed a power function $P \propto v^2$, where $v$ is the voltage. They considered the circuit delay in their model. They demonstrated the theorem with an example plot but did not show the complete proof.

**Energy-based compiler optimization.** Loop fusion has been long studied for improving locality and parallelism. Both benefit energy because they reduce program demand and improve resource utilization [20, 33]. The previous energy studies used only traditional loop-fusion techniques and did not observe a significant performance or energy impact in real benchmarks. They did not consider the balancing effect of loop fusion nor DVS processors.

Compiler techniques have been studied for energy-based code generation, instruction scheduling, and software pipelining. In software pipelining, Yang et al. recently defined balance to be the pair-wise power variation between instructions [32], which is different from our concept of overall variation. Most energy-specific transformations, including that of Yang et al., are applied at the instruction level for a single basic block or the innermost loop body and are targeted toward a conventional processor. Our technique transforms multiple loops at the source level and exploits a unique opportunity made possible by DVS machines.

## 6. Summary

Loop fusion is an important optimization for scientific applications. It has previously been studied as a means of improving performance via reductions in dynamic instruction count and cache miss rate. In this paper we have reconsidered fusion from an energy point of view, and have explored its connection to the concept of program *balance*—of smoothness in demand for processor and memory resources.

By merging program phases, loop fusion tends to even out fluctuations in the instruction mix, allowing the compiler and processor to do a better job of pipeline packing. By moving uses of the same data closer together in time, fusion also tends to reduce the total number of cache misses and the cache miss rate. Improvements in pipeline packing and caching, in turn, tend to increase average processor power, with the result that fusion tends to save more time than it does energy on a conventional superscalar processor. On a processor with dynamic voltage scaling (DVS), however, fusion increases opportunities to slow down the processor in rate-based, soft real-time, memory-bound, or I/O-bound computations, thereby saving extra energy. Moreover the energy savings per percentage of execution slowdown is generally greater in the fused program than it would be in the original, because the fused program's power is higher. As shown in theory in Section 2 and in practice in Section 4.1, fusion can save energy even when it does not save time: increases in program balance always save energy on a DVS processor when time is held constant.

In a related contribution, we presented a methodology in Section 3.4 that enables us to "tease apart" the various factors that contribute to fusion-induced time and energy savings, attributing them to changes in dynamic instruction count and in the effectiveness of caching, branch prediction, pipeline utilization, and DVS. This methodology could, we believe, be used to understand the effect of a wide variety of program transformations. Our current results confirm that fusion tends to reduce the effectiveness of DVS when run time is reduced to the maximum possible extent, but that it introduces opportunities to save dramatic amounts of energy when some of the potential savings in run time is devoted to frequency reduction instead. For our six real-world benchmarks, we demonstrated energy savings ranging from 7–40%, with run times no more than 1% slower—and as much as 17% faster—than those of the original programs.

## Acknowledgments

Tao Li participated at the beginning of this work and gave the first proof of the basic theorem. Michael Huang suggested the use of Jensen's theorem as well as using standard deviation as a measurement.

## References

[1] J. R. Allen, D. Callahan, and K. Kennedy. Automatic Decomposition of Scientifi c Programs for Parallel Execution. In *Proc., 14th Annual ACM Symp. on Principles of Programming Languages*, Jan. 1987.

[2] J. R. Allen and K. Kennedy. Vector Register Allocation. *IEEE Trans. on Computers*, Oct. 1992.

[3] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers, Oct. 2001.

[4] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A Framework for Architectural-Level Analysis and Optimization. In *Proc., 27th Intl. Symp. on Computer Architecture*, June 2000.

[5] T. Burd and R. Brodersen. Processor Design for Portable Systems. *Journal of LSI Signal Processing*, Aug./Sept. 1996.

[6] D. Burger and T. Austin. The SimpleScalar Tool Set, Version 2.0. Technical Report 1342, Dept. of Computer Science, University of Wisconsin-Madison, June 1997.

[7] D. Callahan, J. Cocke, and K. Kennedy. Estimating Interlock and Improving Balance for Pipelined Machines. *Journal of Parallel and Distributed Computing*, Aug. 1988.

[8] S. Carr and K. Kennedy. Improving the Ratio of Memory Operations to Floating-Point Operations in Loops. *ACM Trans. on Programming Languages and Systems*, Nov. 1994.

[9] S. Carr, K. S. McKinley, and C.-W. Tseng. Compiler Optimizations for Improving Data Locality. In *Proc., 6th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Oct. 1994.

[10] L. T. Clark. Circuit Design of XScale$^{TM}$ Microprocessors. In *2001 Symp. on VLSI Circuits, Short Course on Physical Design for Low-Power and High-Performance Microprocessor Circuits*, June 2001.

[11] C. Ding and K. Kennedy. Resource Constrained Loop Fusion. Unpublished manuscript, Oct. 2000.

[12] C. Ding and K. Kennedy. Improving Effective Bandwidth through Compiler Enhancement of Global Cache Reuse. *Journal of Parallel and Distributed Computing*, Jan. 2004.

[13] M. Fleischmann. Crusoe Power Management – Reducing the Operating Power with LongRun. In *Proc., 12th HOT CHIPS Symp.*, Aug. 2000.

[14] G. Gao, R. Olsen, V. Sarkar, and R. Thekkath. Collective Loop Fusion for Array Contraction. In *Proc., 5th Workshop on Languages and Compilers for Parallel Computing*, Aug. 1992.

[15] C.-H. Hsu and U. Kermer. The Design, Implementation and Evaluation of a Compiler Algorithm for CPU Energy Reduction. In *Proc., ACM SIGPLAN Conf. on Programming Language Design and Implementation*, June 2003.

[16] C.-H. Hsu, U. Kremer, and M. Hsiao. Compiler-Directed Dynamic Frequency and Voltage Scaling. In *Proc., Workshop on Power-Aware Computer Systems*, Nov. 2000.

[17] T. Ishihara and H. Yasuura. Voltage Scheduling Problem for Dynamically Variable Voltage Processors. In *Proc., Intl. Symp. on Low Power Electronics and Design*, Aug. 1998.

[18] A. Iyer and D. Marculescu. Power-Performance Evaluation of Globally Asynchronous, Locally Synchronous Processors. In *Proc., 29th Intl. Symp. on Computer Architecture*, May 2002.

[19] J. Jensen. Sur les Fonctions Convexes et les inégalités Entre les Valeurs Moyennes. *ACTA Math.*, 1906.

[20] M. T. Kandemir, N. Vijaykrishnan, M. J. Irwin, and W. Ye. Influence of Compiler Optimizations on System Power. In *Proc., 37th Conf. on Design Automation*, June 2000.

[21] K. Kennedy and K. S. McKinley. Typed Fusion with Applications to Parallel and Sequential Code Generation. Technical Report TR93-208, Dept. of Computer Science, Rice University, Aug. 1993. (also available as CRPC-TR94370).

[22] T. Li and C. Ding. Instruction Balance and its Relation to Program Energy Consumption. In *Proc., Intl. Workshop on Languages and Compilers for Parallel Computing*, Aug. 2001.

[23] G. Magklis, M. L. Scott, G. Semeraro, D. H. Albonesi, and S. Dropsho. Profile-based Dynamic Voltage and Frequency Scaling for a Multiple Clock Domain Processor. In *Proc., 30th Intl. Symp. on Computer Architecture*, June 2003.

[24] J. Ng, D. Kulkarni, W. Li, R. Cox, and S. Bobholz. Inter-Procedural Loop Fusion, Array Contraction and Rotation. In *Proc., Intl. Conf. on Parallel Architectures and Compilation Techniques*, Sept. 2003.

[25] V. S. Pingali, S. A. McKee, W. C. Hsieh, and J. B. Carter. Computation Regrouping: Restructuring Programs for Temporal Data Cache Locality. In *Intl. Conf. on Supercomputing*, June 2002.

[26] G. Semeraro, D. H. Albonesi, S. G. Dropsho, G. Magklis, S. Dwarkadas, and M. L. Scott. Dynamic Frequency and Voltage Control for a Multiple Clock Domain Microarchitecture. In *Proc., 35th Intl. Symp. on Microarchitecture*, Nov. 2002.

[27] G. Semeraro, G. Magklis, 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. In *Proc., 8th Intl. Symp. on High-Performance Computer Architecture*, Feb. 2002.

[28] B. So, M. W. Hall, and P. C. Diniz. A Compiler Approach to Fast Hardware Design Space Exploration in FPGA-based Systems. In *Proc., ACM SIGPLAN Conf. on Programming Language Design and Implementation*, June 2002.

[29] M. M. Strout, L. Carter, and J. Ferrante. Compile-time Composition of Run-time Data and Iteration Reorderings. In *Proc., ACM SIGPLAN Conf. on Programming Language Design and Implementation*, June 2003.

[30] L. Wang, W. Tembe, and S. Pande. A Framework for Loop Distribution on Limited Memory Processors. In *Proc., Intl. Conf. on Compiler Construction*, Mar. 2000.

[31] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers*. PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, Oct. 1982.

[32] H. Yang, G. Gao, C. Leung, R. Govindarajan, and H. Wu. On Achieving Balanced Power Consumption in Software Pipelined Loops. In *Proc., Intl. Conf. on Compilers, Architecture, and Synthesis for Embedded Systems*, Oct. 2002.

[33] H. Yang, G. R. Gao, A. Marquez, G. Cai, and Z. Hu. Power and Energy Impact by Loop Transformations. In *Proc., Workshop on Compilers and Operating Systems for Low Power*, Sept. 2001.

[34] F. Yao, A. Demers, and S. Shenker. A Scheduling Model for Reduced CPU Energy. In *Proc., IEEE Symp. on Foundations of Computer Science*, Oct. 1995.