Aggregating Processor Free Time for Energy Reduction*

Aviral Shrivastava† Eugene Earlie‡ Nikil Dutt† Alex Nicolau†

aviral@ics.uci.edu eugene.earlie@intel.com dutt@ics.uci.edu nicolau@ics.uci.edu

Center for Embedded Computer Systems†
School of Information and Computer Science
University of California, Irvine, CA 92697

Strategic CAD Labs‡
Intel Corporation,
Hudson, MA, 01749

ABSTRACT

Even after carefully tuning the memory characteristics to the application properties and the processor speed, during the execution of real applications there are times when the processor stalls, waiting for data from the memory. Processor stall can be used to increase the throughput by temporarily switching to a different thread of execution, or reduce the power and energy consumption by temporarily switching the processor to low-power mode. However, any such technique has a performance overhead in terms of switching time. Even though over the execution of an application the processor is stalled for a considerable amount of time, each stall duration is too small to profitably perform any state switch. In this paper, we present code transformations to aggregate processor free time. Our experiments on the Intel XScale and Stream kernels show that up to 50,000 processor cycles can be aggregated, and used to profitably switch the processor to low-power mode. We further show that our code transformations can switch the processor to low-power mode for up to 75% of kernel runtime, achieving up to 18% of processor energy savings on multimedia applications. Our technique requires minimal architectural modifications and incurs negligible (< 1%) performance loss.

Categories and Subject Descriptors

C.3 [**Computer Systems Organization**]: Special-purpose and Application-based systems, Real-time and embedded systems; D.3.4 [**Software**]: Programming Languages, Processors [Compilers, Code generation, Optimizations]

General Terms

Algorithms, Performance, Design, Experimentation

*This work was partially funded by grants from Intel Corporation, UC Micro(03-028), and SRC contract 2003-HJ-1111

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CODES+ISSS'05, Sept. 19–21, 2005, Jersey City, New Jersey, USA.
Copyright 2005 ACM 1-59593-161-9/05/0009 ...$5.00.

Keywords

Energy Reduction, Embedded Systems, Processor Free Time, Aggregation, Code Transformation, Clock Gating

1. INTRODUCTION

Memory customization is one of the most important steps in embedded processor design because of its significant impact on the system performance, cost and power consumption. Memory characteristics (like latency, bandwidth, etc.) are numerous and complex, but they should be matched carefully to the application requirements and the processor computation speed. On one hand, processor stalls (processor waiting for data from memory) should be minimized, while on the other hand, idle memory cycles (memory waiting for requests from processor) should be reduced. A lot of time and effort of experienced designers is invested in tuning the memory characteristics of embedded systems to target application behavior and processor computation.

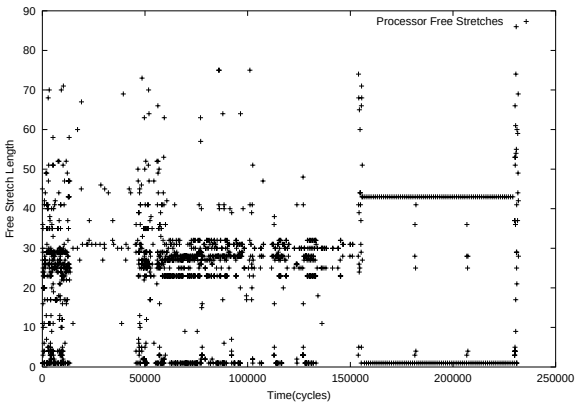


Figure 1: Length of Processor free stretches

However even after such careful design of the memory system there are times during the execution of real-life applications when the processor is stalled, and there are times when the memory is idle. We define a *processor free stretch* to be a sequence of contiguous cycles when the processor is stalled. Figure 1 plots the lengths of processor free stretches over the execution of the *qsort* application from the MiBench benchmark suite[6] running on the Intel XScale [7]. Very small processor free stretches (1-2 cycles) represent processor stalls due to pipeline hazards (e.g. Read After Write

hazard). A lot of free stretches are approximately 30 cycles in length. This reflects the fact that the memory latency is about 30 cycles. An important observation from this graph is that although the processor is stalled for a considerable time (approximately 30% of the total program execution time) the length of each processor free stretch is small. The average length of a processor free stretch is 4 cycles, and all of them are less than 100 cycles.

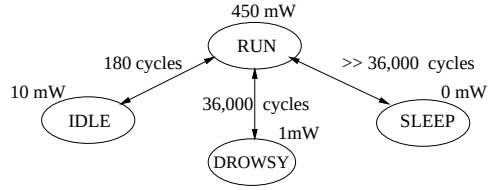


Figure 2: Power State Machine of XScale

A processor free stretch is an opportunity for optimization. System throughput and energy can be improved by temporarily switching to a different thread of execution [11]. The energy consumption of the system may be reduced by switching the processor to a low power state. Figure 2 shows the power state machine [2] of the Intel XScale. In the default mode of operation, the XScale is in RUN state, consuming 450mW. XScale has three low power states: IDLE, DROWSY, and SLEEP. In the IDLE state the clock to the processor is gated, and the processor consumes only 10mW. However, it takes 180 processor cycles to switch to and back from the IDLE state. In the DROWSY state the clock generation circuit is turned off, while in the SLEEP state the processor is shut down. DROWSY and SLEEP states have much reduced power consumption, but at an increased cost of state transition. The break-even processor free stretch for a transition to IDLE state is $180 \times 2 = 360$ cycles. This implies that the processor free stretch should be more than 360 cycles to profitably switch the processor to IDLE state. Clearly existing processor free stretches are not large enough to achieve power savings by switching to a low-power state. It is important to note that this is in-spite of the fact that the total processor stall duration is quite large.

Traditional assumption has been that power optimization opportunities by switching the processor to low-power state can only be found at the operating system level (inter-process), and not at the compiler-level (intra-process). Consequently a lot of research has been done to develop power optimization techniques, that operate application level granularity, and are controlled by the operating system. To the best of our knowledge there has been no research to aggregate processor free times within an application to reduce the power/energy consumption of the processor.

In this paper we address this traditional oversight. We present code transformations to aggregate processor free times and obtain large chunks of processor free time, in which the processor can be profitably switched to a low-power state. Our experimental results on Stream kernels running on an Intel XScale show that our technique can aggregate processor free times to up to 50,000 processor cycles and thereby switching the processor to low-power state for up to 75% of kernel execution time. Further, we show that for real-life multimedia applications our technique achieves up to 18% savings in processor energy. Minimal architectural support is needed for our approach and it incurs negligible performance ($< 1\%$) impact.

2. RELATED WORK

Prefetching can be thought of as a technique to aggregate memory activity. [12] presents an exhaustive survey of hardware, software, and cooperative prefetching mechanisms. Processor free cycles can be used to increase the throughput of systems by switching to a different thread of execution [11]. They can also be used to reduce the power consumption of a processor by switching it to a low-power mode [9]. Clock gating, power gating, frequency scaling and voltage scaling provide architectural support for such low-power states [9].

Several power/energy reduction techniques work at the operating system level by switching the whole processor to a low-power mode. The operating system estimates the processor free time using predictive, adaptive or stochastic techniques [2]. Some power/energy reduction techniques operate at the application level but they control only a very small part of the processor, e.g. multiplier, floating point unit. The compiler statically or hardware dynamically estimates the inactive parts of processor, and switches them to low-power mode [5].

However, there are no existing techniques to switch large parts of processor to a low-power state, during the execution of an application. This is mainly because during the execution of an application, without aggregation the processor rest times are too small to profitably make the transition to low-power state. This is true even though the total stall duration may be quite significant. In this paper we present a technique to aggregate small processor stalls to create a large free chunk of time when the processor is idle, and can be profitably switched to a low-power state.

3. MOTIVATION

Consider the loop expressed in Figure 3(a). In each iteration the next element from two arrays a and b are loaded and their sum is stored into the third array c. The ARM assembly code of the loop generated by GCC is shown in Figure 3(b).

```
for (i=0; i<1000; i++)
  c[i] = a[i] + b[i];
```

(a)

```
1. L: mov   ip, r1, lsl#2
2.    ldr   r2, [r4, ip]
3.    ldr   r3, [r5, ip]
4.    add   r1, r1, #1
5.    cmp   r1, r0
6.    add   r3, r3, r2
7.    str   r3, [r6, ip]
8.    ble   L
```

(b)

Figure 3: Example Loop

Consider a simple hypothetical pipeline model of a processor. In the absence of cache misses every instruction takes one cycle to execute. To model a realistic memory behavior we need a slightly complicated memory model. The memory architecture of the processor is described in Figure 4. We assume there is a request buffer in front of the cache. This makes the cache non-blocking (the processor does not stall on a cache miss). The request latency of the memory (time elapsed between making a request to getting the first word) is 12 cycles. The memory bus is pipelined so that multiple requests can be pending and hide the memory latency. We assume independent request and data bus for increased efficiency of memory bus. The memory bandwidth is 1 word per 3 cycles and the cache line size is 16 bytes.

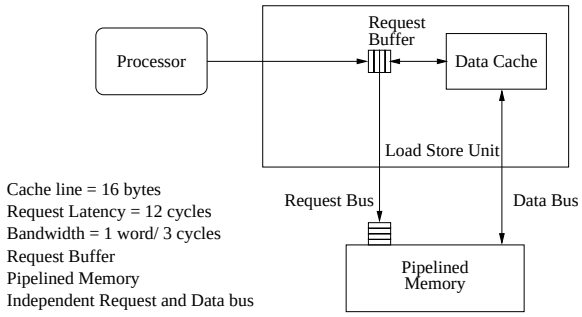


Figure 4: Example Memory Architecture

To simplify the analysis we assume simple loops with sequential array accesses, like the loop in Figure 3. The analysis can be easily generalized later. For such loops it is convenient to think of memory operations per k loop iterations; where k is the number of elements (of the type that are being accessed in the loop) that fit in one cache line. For example the loop in Figure 3 loads integers(4 bytes), thus $k = 16/4 = 4$. Therefore we define an Iteration (with a capital I) as 4 iterations (with a small i). In one Iteration, the loop needs to load 3 lines, one line each from arrays a, b, and c.

The Computation C in a loop is defined to be the number of cycles required to execute one Iteration of the loop if there are no cache misses. For this loop, $C = 8 \times 4 = 32$ cycles. The Memory Latency ML of a loop is defined as the number of cycles required to transfer all the data required in one Iteration in steady state, assuming that the request was made well in advance. In every Iteration in steady state, three lines have to be loaded (one line each from arrays a, b, and c) and one line has to be written back (one out of every three lines, *the one corresponding to array c*, is dirty). Thus $ML = 4 \times 4 \times 3 = 48$ cycles.

For this loop $ML > C$. We call such a loop as memory bound, and there is no way to avoid processor stalls for such a loop. Even if the request for the data is made well in advance, the memory bandwidth is not enough to transfer the required data in time. For loops in which $ML = C$, the memory requirement and computation are matched. The memory system can be said to be tuned for such loops.

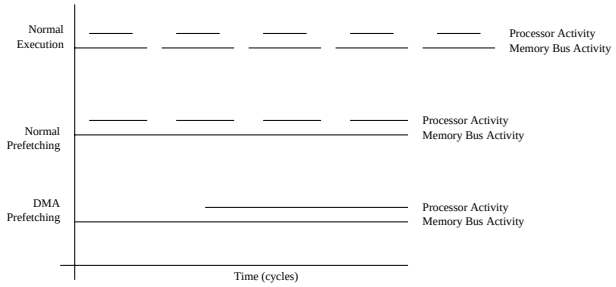


Figure 5: Processor Memory Activity

The top two sets of lines in Figure 5 represent the processor and memory bus activity in a normal processor. The processor executes intermittently. It has to stall at the 6^{th} instruction of each Iteration, and wait for the memory to catch up. The memory activity starts with the request made by the 2^{nd} instruction, but data continues to transfer on the memory bus, even after the processor has stalled. Even before all this data could be transfered, there is a new request from the 2^{nd} instruction of the next Iteration of the loop, and the request latency could not be completely hidden. Thus in a normal processor, both the processor and memory bus activities are intermittent.

Prefetching has the effect of aggregating the memory bus activity. The next two horizontal lines from the top in Figure 5 represent the processor and memory bus activity with prefetching. Each period of processor execution is longer because of the computational overhead of prefetching. However, the memory bus activity is now continuous. The memory bus is one of the most critical resources in several systems, including our example architecture. Improving the utilization of such an important resource is very important. As a result of this, the runtime of the loop decreases.

However, the processor utilization is still intermittent. If we can aggregate the processor utilization, we can achieve an activity graph as shown by the bottom two lines of Figure 5. The plot clearly shows that the processor needs to be active only for a fraction of time of the memory bus activity, the fraction being equal to C/ML. Furthermore the aggregated inactivity window can be large enough to enable profitable switching to low-power mode, and energy reduction achieved. In this paper we present a code transformation and architectural support required to achieve this aggregation.

4. AGGREGATION APPROACH

```
for (i=0; i<1000; i++)
  c[i] = a[i] + b[i];
```

(a)

```
startPrefetch()
for (i1=0; i1<T2; i1+=T)
  setProcWakeup(w)
  procIdleMode()
  for (i2=i1; i2<i1+T; i2++)
    c[i2] = a[i2] + b[i2]
```

(b)

Figure 6: Loop Transformation

Our idea of aggregating processor rest time is very intuitive, and requires a prefetch engine. Figure 6 shows the transformation of the loop shown in Figure 3. For each memory bound loop in the application we find out which arrays and how many lines from each array are required by the loop and the prefetch is started. The loop is first tiled with tile of size T. Inside each tile, the wakeup factor w is set in the prefetch engine to wake up the processor. The processor is then switched to a low-power state. In the low-power state, the processor waits for an interrupt from the prefetch engine. All parts of processor except the load store unit are switched to a low-power state. At appropriate time the prefetch engine generates an interrupt to wake up the processor. The processor wakes up and resumes execution on the prefetched data in the cache. The prefetch engine continues to work even after the processor is awake. The tile size T, and the wake up parameter w are computed so as to maximize the time the processor is in the low-power state.

Note that Figure 6 shows only the transformation of the loop kernel. However to implement this transformation, we need to identify the loop kernel, the epilogue and the prologue of the loop. First we describe the complete loop transformation steps, then we show the calculation of T and w to achieve the maximum energy savings, and finally describe our prefetch engine implementation.

4.1 Code Transformation

1. Identify simple memory bound loops a. C can be estimated by assembly code of loop b. ML can be estimated by source code of loop
2. Break the loop into two parts a. Before steady state b. At steady state
3. Find MLof the first part of loop by profiling Compute wakeup time for first part of loop
4. Find tiling factor and wakeup time for second part of the loop Break away the third part (tail) of the loop
5. Find the wakeup time for the third part of loop

Figure 7: Transformation Steps

The transformation is applied only to memory bound loops. The loop transformation described in Figure 7, converts the loop in Figure 3 into the loop in Figure 8. The first step in the transformation is to estimate the memory latency ML and the computation C of the loop. The ML can be estimated by a simple source code analysis, while C can be estimated by a simple assembly code analysis. We define the steady state of the loop, as when the loop has consumed a cache-full of data. Before steady state, the memory latency of the loop may not be equal to ML. If the data required in an Iteration is already present in the cache, there will be no memory transfers. On the other hand, if the data residing in the cache is dirty, then more writebacks will happen. The loop is therefore broken into two parts, the first one is before steady state ($i = 0; i < T1$), the prologue, and the second one is at steady state ($i = T1; i < N$). For this we need to estimate $T1$, the break point of the loop.

Before steady state, profile information is needed to estimate ML. A cycle accurate simulator is instrumented to count all memory activity in the loop in the first part of the loop. Thus for the prologue, $ML = cache\_misses/T1$. Using ML, C and $T1$, the wake up parameter $w1$ for the first part of the loop is computed.

For the second and the main part of the loop, ML and C are known. The tile size T of the loop needs to be computed. There will be $(N - T1)/T$ tiles in the second part of the loop. However, there may be a tail end of the loop left ($i = ((N - T1)/T) \times T; i < N$). The tail end of the loop, is separated as the third part or epilogue of the loop. In the main kernel, the processor will be switched to low-power mode at the beginning of each tile, and processor wake up factor will be setup in the prefetch engine. The processor wakes up by the interrupt from the prefetch engine, and starts executing. The wake up factor w is computed in such a way that the tile computation ends by the time processor exhausts the prefetched data. The wake up time $w2$ for the epilogue is also estimated.

Currently our analysis of aggregating processor free times is for loops that request consecutive lines from each array. It should be possible to extend the scope of this transformation by employing more sophisticated data analysis techniques. The overhead of the additional instructions inserted by our transformation is negligible ($< 1\%$ of the total runtime).

```
// Set the prefetch engine
1. setPrefetchArray a, N/L
2. setPrefetchArray b, N/L
3. setPrefetchArray b, N/L
4. startPrefetch

// first part of the loop
5. setProcWakeup w1
6. procIdleMode
7. for (i1=0; i1<T1; i1++)
8.    c[i1] = a[i1] + b[i1]

// tile the second part of the loop
9. for (i1=0; i1<T2; i1+=T)
10.   setProcWakeup w
11.   procIdleMode
12.   for (i2=i1; i2<i1+T; i2++)
13.      c[i2] = a[i2] + b[i2]

// tail of the loop
14. setProcWakeup w2
15. procIdleMode
16. for (i1=T2; i1<N; i1++)
17.    c[i1] = a[i1] + b[i1]
```

Figure 8: Fully Transformed Loop

4.2 Computation of Loop breakpoints, Tile size and Wake up Factor

The loop breakpoints, tile size, and wake up time depend on the the loop characteristics, cache parameters, and the location of arrays in memory. For our analysis we assume that C is the computation, and ML is the memory latency of the loop. Also assume that in each Iteration of the loop, r lines need to be read. Further assume that the cache has s sets, and has associativity a.

We first compute the first breakpoint $T1$ of the loop. The steady state starts when the loop has consumed one cache-full of data. In s Iterations, the loop completely uses r lines of each set. Thus it will use the whole cache and a little more in $T1 = a \times s/r + s$ Iterations. To compute the wake up time $w1$ of the processor, given the number of Iterations the loop will execute, the following condition must hold true, $C \times T1 = ML \times (T1 - w1/r)$. This ensures that enough data has been loaded into the cache to perform computation without any cache misses. Thus $w1 = ((ML - C) \times T1 \times r)/ML$.

The wake up time w of the processor inside the tile is computed such that each time we prefetch a cache-full of data, thus $w/r = (a/r) \times s$. The tile size T can then be computed by the following relation, $(C \times T/ML) + w/r = T$. This implies $T = w/r \times ML/(ML - C)$.

The wake up time $w2$ for the third part of the loop can be computed similarly as $w2/r = ((ML - C) \times (N - T2))/ML$.

4.3 Architectural Support

Very little architectural support is needed for the proposed scheme to work. The exact implementation of the prefetch engine as well as its interface may differ for each architecture. Our implementation is shown in Figure 9. A mechanism to set up the prefetch engine is required. The processor needs to specify the start of the arrays, and the number of lines of each array to be prefetched to the prefetch

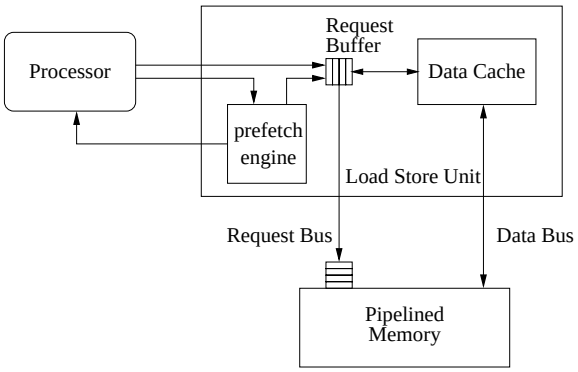


Figure 9: Prefetch Engine

engine. The processor also needs to specify the number of lines to prefetch before waking up the processor (w). A mechanism to start the prefetch engine and a way to switch the processor to a low-power mode is needed. In the low power mode, the clock to the processor(other than the load store unit) can be frozen, and even powered-down. The processor just waits for an interrupt from the prefetch engine. The prefetch engine keeps prefetching by inserting requests in the request buffer. The prefetch engine monitors the request buffer, and it does not let the request buffer be empty. If the request buffer is always full, the activity on the data bus remains uninterrupted. This ensures the maximum usage of data bus by keeping the request buffer non-empty. After the prefetch engine has requested more than w lines from the memory, it should generate an interrupt to wake up the processor. The processor should catch the interrupt and resume execution again. The prefetch engine continues its operation even after waking up the processor. After all the data has been prefetched, the prefetch engine should disengage itself, until it is invoked by the processor again.

5. EXPERIMENTS

To demonstrate the usefulness and efficacy of our approach we perform experiments on our XScale simulator. The simulator has been validated against the 80200 EVB (XScale Evaluation Board) [1] to be accurate within 7% on an average in cycle count measurements. The simulator has been extended to model the prefetch engine. The code transformations have been applied on the best performing code generated by GCC.

5.1 Steady-state Analysis

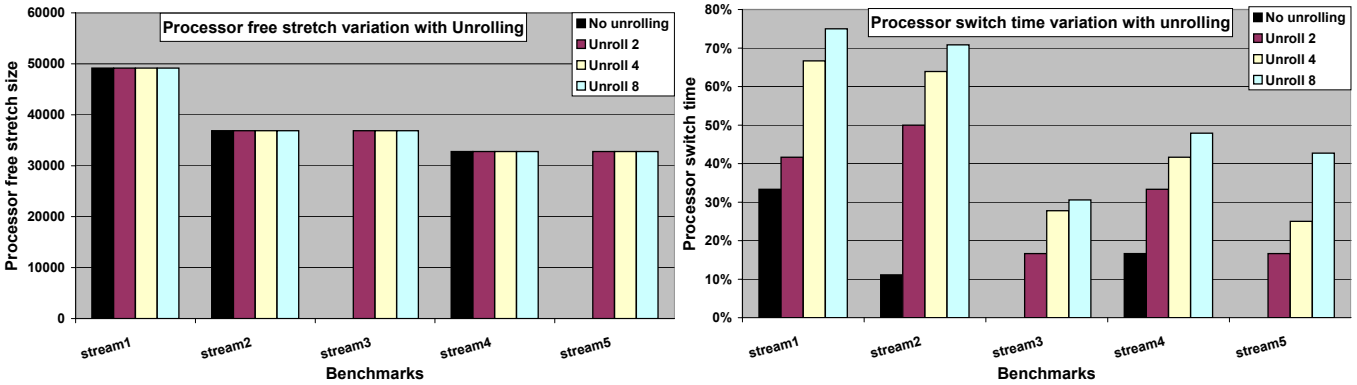
We perform the first set of experiments on kernels from the Stream suite[8]. Stream kernels are widely used to balance the memory bandwidth to the computation speed of processors. The kernels have varying degree of computation to memory requirement (C/ML) ratio. The leftmost (black) bars in Figure 10(a) plot the processor free stretch our aggregation technique could obtain for each kernel in Stream suite. Up to 50,000 processor free cycles can be accumulated. Two of the kernels, *stream3* and *stream5*, do not have black bars. Our technique was unable to aggregate processor free stretches for these two kernels, because $C > ML$ for these kernels. The leftmost (black) bars in Figure 10(b) represent the percentage of total execution time, the processor can be switched to a low-power mode (processor switch time). The processor state switching time (360 cycles for each switch), is considered to be full-power mode. The processor can be switched to a low-power mode for up to 33% of the program execution time. Note again, that the processor cannot be switched to a low-power mode in *stream3* and *stream5*. This shows that our technique is able to aggregate processor free times into large chunks, so that it becomes profitable to switch the processor to a low-power mode. Furthermore, our results show that the processor can be switched to low-power mode for a significant amount of kernel run-time.

5.2 Effect of Loop Unrolling

The processor free time our technique is able to aggregate is proportional to the ratio, C/ML. Optimizations that improve this ratio improve the aggregation results. Loop unrolling is a popular optimization that reduces the computation per Iteration (C) of a loop. In this subsection we analyze the impact of loop unrolling on the processor free stretch and the processor switch time obtained. Figure 10(a) shows that the processor free stretch size obtained is not affected by unrolling. As discussed in Section 4, the ability to accumulate processor free time is dependent on whether $ML > C$, but if it is possible, the size of the processor free stretch obtained is proportional to ML. It does not increase by reducing C. Figure 10(b) show plots the processor switch time percentage obtained for the Stream kernels for different unroll factors. The first observation is that after unrolling the processor can find switching opportunities even for kernels *stream3* and *stream5*, which were earlier deemed unprofitable. The second important obeservation is that unrolling increases the processor switch time percentage for the kernels. The processor can now be switched to a low-power mode for up to 75% of kernel run-time. Thus unrolling improves the applicability and efficacy of our aggregation technique. It is worth mentioning here that hand-generated assembly code has even better C/ML ratio. Hand generated assembly uses much fewer instructions to code the same functionality (thereby reducing C) , but the ML remains the same. Thus the aggregation techniques works even better on hand-generated assembly code.

5.3 Energy Saving Estimation

In this subsection we estimate the energy consumption of the processor with and without applying our aggregation technique. We develop simple and conservative state-based processor energy models. We show that even with such conservative energy models, our technique achieves significant processor energy reduction. In the normal RUN mode, operating at 600 MHz, the XScale consumes 450 mW. In the STALL mode, the XScale consumes 25% of 450 mW = 112.5 mW [3, 7, 4]. In the aggregated processor free time, we switch the processor to our low-power mode called MY_IDLE. In the normal IDLE mode of XScale, the clock to the processor and memory are gated, and XScale consumes 10 mW. However in our idle state, MY_IDLE, the memory clock needs to be on, the prefetch engine is on, and we need to keep performing writes in the cache. Clock consumes about 20% power in the XScale, i.e. 90 mW. The clock power is divided into processor clock power and memory clock power. We assume that processor clock is 6 times faster than memory clock. The capacitive load on the processor clock is much more than the capacitive load on the memory clock. We conservatively assume same capacitive



(a) Variation of processor free stretch size with Unrolling

(b) Variation of processor switch time with Unrolling

Figure 10: Processor free stretch size and switch time variation with Unroll factor on Stream benchmarks

load, therefore memory clock consumes $90/7 = 13$ mW. Caches in XScale consume about 25% power, i.e. 112.5 mW. However this is divided into data cache power and the instruction cache power. In XScale the instruction cache and data cache have the same architectural parameters. However on an average the instruction cache is accessed every cycle, while the data cache is accessed every 3 cycles. Thus energy consumed by the data cache is $112.5/4 = 28$ mW. We synthesized the prefetch engine using design-compiler of Synopsys-2001.10 on a 0.8μ library, and estimated its power consumption using Synopsys power-estimate[10]. The power consumption, scaled to the XScale process technology (0.18μ) is less than 1 mW. Thus in MY_IDLE state, processor consumes $13+28+1 = 42$ mW. However for our experiments, we make a further conservative estimate, and assume processor power consumption to be 50 mW. We augmented the cycle accurate simulator with the power of RUN state, STALL state and MY_IDLE state. Depending on the current state of the processor, the corresponding state power is added and the total energy consumption computed. Figure 11 shows the even with such a conservative estimate, up to 18% processor energy savings can be obtained by our processor free time aggregation technique on multimedia applications running on XScale.

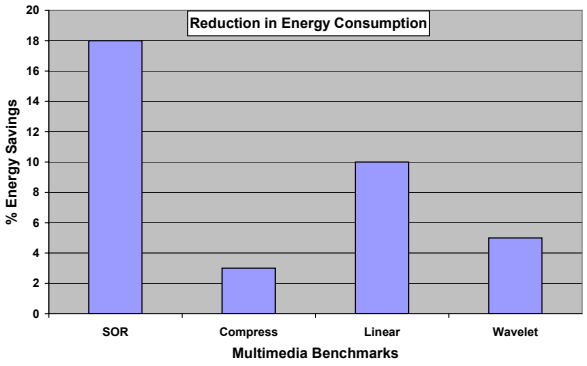


Figure 11: Energy savings on multimedia apps.

6. SUMMARY

Although the processor is stalled for a significant duration over program execution, each stall duration is too small to profitably perform any performance and energy optimization. In this paper we presented a code transformation technique that can aggregate up to 50,000 processor free cycles on the XScale over Stream kernels. The aggregated processor free stretch cycles can be used to profitably switch the processor to low-power mode for upto 75% of kernel runtime. Furthermore we have shown that our technique can save up to 18% of system-wide energy consumption on real-life multimedia benchmarks. The code size, performance penalties, and energy overhead of our technique are negligible ($< 1\%$), demonstrating the feasability of our approach. As yet our analysis can handle only simple loops. We are working to extend the applicability of our technique to more general loops, and also to non-loop regions of code.

7. REFERENCES

[1] Intel 80200 processor based on intel xscale microarchitecture.

[2] L. Benini, A. Bogliolo, and G. D. Micheli. A survey of design techniques for system-level dynamic power management. *IEEE Transactions on VLSI Systems*, 8(3):299–316, 2000.

[3] L. T. Clark, E. J. Hoffman, M. Biyani, Y. Liao, S. Strazdus, M. Morrow, K. E. Velarde, and M. A. Yarch. An embedded 32-b microprocessor core for low-power and high-performance applications. *IEEE Journal of Solid State Circuits*, 36(11):1599–1608, 2001.

[4] G. Contreras, M. Martonosi, J. Peng, R. Ju, and G. Y. Lueh. XTREM: A power simulator for the intel xscale core. In *LCTES*, 2004.

[5] M. K. Gowan, L. L. Biro, and D. B. Jackson. Power considerations in the design of the alpha 21264 microprocessor. In *Design Automation Conference*, pages 726–731, 1998.

[6] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *IEEE Workshop in workload characterization*, 2001.

[7] Intel Corporation, http://www.intel.com/design/intelxscale/273473.htm. *Intel XScale(R) Core: Developer's Manual.*

[8] J. D. McCuplin. Memory bandwidth and machine balance in current high performance computers. *Newsletter of IEEE Computer Architecture Technical Committee*, 1:19–25, 1995.

[9] J. M. Rabaey and M. Pedram. Low power design methodologies. In *Kluwer*, 1996.

[10] Synopsys. *Synopsys Design Compiler.* http://www.synopsys.com/products/logic/design_compiler.html, 2001.

[11] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: maximizing on-chip parallelism. *SIGARCH Comput. Archit. News*, 23(2):392–403, 1995.

[12] S. P. Vanderwiel and D. J. Lilja. Data prefetch mechanisms. *ACM Computing Surveys (CSUR)*, 32(2):174–199, 2000.