Static Analysis of Processor Stall Cycle Aggregation *

Jongeun Lee
Jongeun.Lee@asu.edu

Aviral Shrivastava
Aviral.Shrivastava@asu.edu

Department of Computer Science and Engineering
Arizona State University
Tempe, AZ 85281

ABSTRACT

Processor Idle Cycle Aggregation (PICA) is a promising approach for low power execution of processors, in which small memory stalls are aggregated to create a large one, and the processor is switched to low-power mode in it. We extend the previous proposed approach in two dimensions. i) We develop static analysis for the PICA technique and present optimum parameters for five common types of loops based on steady-state analysis. ii) We show that software only control is unable to guarantee its correctness in a varying runtime environment, potentially causing deadlocks. We enhance the robustness of PICA with minimal hardware extension, ensuring correct execution for any loops and parameters, which greatly facilitates exploration based parameter optimization. The combined use of our static analysis and exploration based fine-tuning makes the PICA technique applicable, to **any** memory-bound loop, with energy reduction. We validate our analytical models against simulation based optimization and also show through our experiments on embedded application benchmarks, that our technique can be applied to a wide range of loops with average 20% energy reductions compared to executions without PICA.

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—*Code generation, Compilers, Optimization*

General Terms

Algorithm, Design, Experimentation, Performance

Keywords

Low power, code transformation, embedded systems, memory bound loops, processor free time, stall cycle aggregation

*The authors would like to thank Eugene Earlie, Intel Inc., and Kalyan Basu, Microsoft for their insightful ideas. We would also like to thank Microsoft Research and Science Foundation of Arizona (SFAz) for their support.

CODES+ISSS'08, October 19–24, 2008, Atlanta, Georgia, USA.

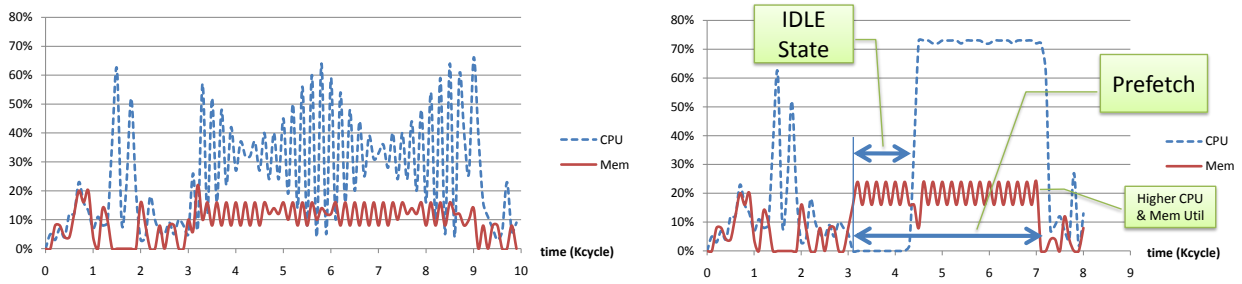
1. INTRODUCTION

Power may well be the single most important concern in the design of battery-operated handheld devices. The battery is typically the prime determinant of the weight, volume, shape, size, charging time, charging frequency, and ultimately the usability of the portable system. Consequently decreasing the power consumption of embedded processors is an important research concern.

Many power reduction techniques fundamentally save power when the full performance is not needed. While Dynamic Voltage and Frequency Scaling (DVFS) techniques [1] attempt to discover execution intervals when the processor can be slowed down, Dynamic Power Management (DPM) implemented using clock gating, power gating, etc. [2] attempt to discover opportunities to stop the processor, or parts thereof, without hurting the performance. Such DPMs are realized in processors in the form of power states; e.g., the Intel XScale processor has 3 low-power states, IDLE, DROWSY, and SLEEP. However, the time to switch to even the shallowest low power mode, IDLE, is 180 processor cycles and additional 180 processor cycles to come back from it. Interestingly, when the Intel XScale is executing, although the processor experiences memory stalls almost 30% of the time (IPC = 0.7), practically no stall is more than 360 processor cycles. Thus while the total stall time is significant, each stall duration is too small to save power by switching the processor to low-power state. Consequently, most previous techniques attempt to switch the processor to low-power states in between applications, or when task deadlines are known beforehand such as in real-time systems.

We earlier proposed a technique [3] to collect several small stalls together to create a large stall in memory-bound loops. Processor power can be saved by switching the processor to low-power mode during the large stall. The aggregation technique is a hardware-software cooperative approach in which the compiler analyzes the application to find out what needs to be prefetched. It delegates the task of large scale prefetching to a programmable prefetch engine and switches to low-power mode. The prefetch engine brings data from the memory to the cache on behalf of the processor, and it wakes up the processor at a pre-determined time. The processor wakes up and operates on the data in the cache without any memory stalls. Figure 1 plots the computation and data transfer rates (number of useful cycles per every 100 cycles) for a simple loop, before and after applying processor aggregation. In (b), soon after cycle 3000 prefetch is activated and the processor is switched to the IDLE state for about 1000 cycles. Once the processor is woken up it runs much faster because there is no memory stall. Because of large scale prefetching, the data transfer rate is higher in the processor aggregation case, and there is a runtime reduction of about 20%.

While processor stall cycle aggregation can reduce processor



(a) Before aggregation

(b) After aggregation

Figure 1: CPU and memory utilization (Percentage of cycles when useful work is done).

power consumption without performance penalty for memory-bound loops, the previous proposal has several limitations. In this paper, we significantly enhance our previously proposed aggregation technique, both in terms of applicability and energy efficiency, and call it PICA (Process Idle Cycle Aggregation).

(i) Previous aggregation approach depended on the compiler to estimate key parameters of aggregation, i.e., wakeup time w, which is the number of lines that the prefetch engine should fetch before waking up the processor. Due to the non-determinism in the memory speed the processor may not wakeup from the low-power state, resulting in a deadlock. In this paper we enhance the aggregation mechanism with hardware deadlock prevention mechanism. A major advantage of our deadlock-free PICA is that it enables us to determine the PICA parameters by simulation, making PICA applicable to any loop, with power reductions on memory-bound loops.

(ii) Static code analysis to determine aggregation parameters presented in the previous proposal worked only for a specific kind of loop. We profile several important applications, classify the kinds of loop present, and develop code analysis techniques for them. This makes PICA applicable to a wide variety of important applications.

(iii) Finally, we propose to find the starting values of the PICA parameter exploration through static compiler analysis, and then fine tune it by simulations to achieve the maximum energy savings. PICA achieves average 20% power reduction on a wide variety of loops.

2. RELATED WORK

DVFS [1, 4] implementation requires a voltage regulator which is fundamentally different from a standard voltage regulator because it must also change the operating voltage for a new clock frequency [5]. This and more considerations result in high transition overhead for DVFS. The Intel XScale processor has frequency switching time of $20\mu s$ [6, 7]. Consequently, in order to hide the penalty of voltage regulation, DVFS is applied at the context switching granularity, typically by the operating system.

In contrast, DPM implemented using clock gating, power gating, body biasing, etc. [2, 8] has relatively low transition overhead. Stopping parts of the processor, such as power gating of functional units, has a penalty of about 10 processor cycles, and therefore can be controlled by the compiler. However, stopping the whole processor has much higher overhead, thus not being exploited by compilers.

Our PICA approach collects small stalls and creates a long stall so that the processors can be profitably switched to low-power. This technique is primarily based on large scale prefetching. Software prefetching techniques [9] augment the application code with *fetch* instructions, which will bring additional lines from the memory.

```
1: for ( i = 0; i < N; i++ ) do
2:     Any statements involving i
3: end for
```

(a) Original loop

```
1: setPrefetchArray addr, stride, #lines, ···
2: startPrefetch
3: for (j = 0; j < N; j+=T) do
4:     procIdleMode w
5:     M = min(j + T, N)
6:     for ( i = j; i < M; i++ ) do
7:         Same statements involving i
8:     end for
9: end for
```

(b) PICA-transformed loop

Figure 2: Simplified PICA transformation.

Similarly, hardware prefetching mechanisms [10] predict what will be needed next in the hardware itself, and request the predicted data. In either case, prefetching data that will not be used, i.e., incorrect prediction increases cache pollution and degrades power and performance.

While the issue of when to prefetch is simple in small scale prefetching mechanisms, for large scale prefetches proper scheduling is very important so as not to overwrite still-to-be-used data in the cache. While scratch pad management techniques [11, 12, 13] attempt to solve this problem, they do not consider cache eviction and write-back. In this paper, we present steady-state analysis of the data in the cache for several kinds of loops, and increase the applicability and effectiveness of PICA approach by answering the question of "when to prefetch."

3. DEADLOCK-FREE PICA

Figure 2 shows the loop transformation that is required to perform processor stall aggregation. While the prefetch of the required data is initiated at the beginning of the loop, the original loop, that ran from $i = 0$ to $i = N$, has to be tiled into "tiles" of size T. Within each tile, in line 4, the processor is first put into the low-power mode with parameter w. The prefetch continues to transfer data between the cache and the memory, and after it has made a request for w lines, it wakes up the processor. When the processor wakes up, it will work on the data present in the cache. But since it will *consume* data faster than memory can *produce*, T is computed so that the tile ends just when all the data in the cache is used up, and the processor will miss in the cache if it continues anymore. w is computed under the constraint that memory should not overwrite the still-to-be-used data in the cache. Both these parameters are

Table 1: Modified prefetch engine behavior

Instruction	Description
setPrefetchArray	Add to $counter1$ the number of lines to request
startPrefetch	Start $counter1$ (decrement it by one for every line completed)
procIdleMode w	Set $counter2$ to w and put processor into sleep mode only if $w \leq counter1$, otherwise do nothing. Also start $counter2$ (decrement it by one for every line completed; upon reaching zero wake up the processor)

Table 2: Loop classification

Type	AM	RM	SS	Example
1	multi	single	all ref.'s	$A[i] + B[i] + C[i]$
2	multi	single	none	$A[i] + B[2i]$
3	single	multi	all ref.'s	$A[i] + A[i + 10]$
4	multi	multi	all ref.'s	$A[i] + A[i + 10] + B[i] + B[i + 20]$
5	multi	multi	all ref.'s to same array	$A[i] + A[i + 10] + B[2i] + B[2i + 30]$
6	single	multi	none	$A[i] + A[2i]$
7	multi	multi	none	$A[i] + A[2i] + B[i + 10] + B[3i + 15]$

estimated by the compiler. However, due to indeterministic cache state, variable memory speed, or incorrect estimation of w, it is possible that during the time when the processor is awake, it may fetch more lines than estimated by the compiler. Note that the number of lines fetched while the processor is in low-power mode is controlled by the parameter w, but the number of lines fetched when the processor is awake is not controlled. This mismatch in the actual memory speed and the compiler estimated memory speed may result in more lines being fetched than what compiler estimated. If less then w lines remain to be fetched when the processor goes into the idle mode, then it will not be able to get out of the idle mode, as the prefetch engine will not generate w requests.

To avoid deadlocks we add a simple hardware logic to the prefetch engine, which checks whether there are enough lines to fetch before putting the processor to the idle mode. This can be implemented with little overhead using a counter for the number of the remaining line requests. The prefetch engine commands have to be modified accordingly, as summarized in Table 1. (Previously separate commands, setProcWakeup and procIdleMode, are merged into one, procIdleMode.) In addition to making PICA robust, this enhancement greatly improves the applicability of our technique. In the previous approach, the compiler was responsible to estimate w and T, so as to avoid the deadlock. Pessimistic estimates by the compiler to guarantee correctness resulted in lower energy reductions. Since PICA avoid deadlocks, it facilitates exploration-based parameter optimization. We can aggressively explore the parameter space and find the optimal PICA parameters w and T for *any* memory-bound loop to maximize the energy reduction. This greatly enhances the applicability of PICA on complex programs, which may be out of the reach of traditional compiler analysis.

4. ANALYTICAL PICA OPTIMIZATION

Although PICA parameters can be determined through an exploration based approach, the exploration space is quite large. The upper bound on T is the number of iterations. An upper bound on w is the minimum of total number of lines in the cache and the total number of lines programmed to be prefetched. Since exploration based PICA parameter optimization may be time consuming, it is valuable to develop analytical techniques to optimize PICA parameters. We present our analysis on steady-state optimal PICA parameters for common types of loops.

4.1 Loop Classification

We studied several multimedia and DSP applications to find out all the memory bound loops. Most of the loops with compile-time deterministic access pattern fall into the 7 types listed in Table 2, which classifies loops based on the multiplicity of arrays in the loop (AM), the multiplicity of references to each array (RM), and whether (or which) references have the same speed (SS).

Type 1 is the simplest and a generalization of the only case addressed in our earlier work [3]. Types 2 and 5 allow different arrays to be accessed at different speeds.[1] The last two types allow references to the same array to have different speeds. In this case accurate steady-state analysis becomes very difficult, since the distances[2] between the references also change over time. In fact there are only two modes—when the distance is short enough the references can be considered to be *overlapping*, but when the distance is long enough they behave just like separate references to different arrays. And because the distance changes over time, we cannot determine the parameter value that is optimal at all times throughout the iterations.

We have analytically computed the PICA parameters w and T for the first five types of loops. Due to the lack of space we present only the analysis of type 4 loops, which are moderately complex allowing multiple arrays with multiple references and are more general than types 1 and 3.

4.2 Input

We consider innermost loops only, since prefetching for nested loops may require much more complex prefetch functionality than simple stride. Multi-dimensional arrays can be treated in our work as they can be easily converted to single-dimensional arrays accessed with strides. In the steady state a write reference is equal to two read references at the same speed in terms of data transfer. Thus we consider only read accesses in the analysis. For static analysis we assume fully-associative caches and FIFO (First-In-First-Out) replacement policy.

An array may be accessed by one or more references. We consider references that are affine linear expressions of the iterator. The production rate p_i of a reference is the average number of cache lines newly needed by the reference per iteration. This is the rate at which the prefetch engine needs to produce data ideally. For example, if iterator i is incremented by one, A is an 4-byte integer array, and the data cache has 32-byte lines, a reference $A[i + k_1]$ will need a new cache line every eight ($= 32/4$) iterations. Therefore the production rate of this reference is $p_1 = 1/8$. Another reference $A[a\,i + k_2]$, under the same conditions, will need a new cache line every $32/(4a)$ iterations if $a \leq 8$. If $a > 8$, this reference will need a new cache line every iteration. Therefore the production rate for this reference is $p_2 = \min(a/8, 1)$. Constants, k_1 and k_2, do not affect the production rates.

4.3 Array-Iteration Diagram

We use *array-iteration diagram* to capture the data access pattern of references in a loop, in space and time, as illustrated in Figure 3. The vertical axis represents the elements of arrays. The horizontal axis represents time in terms of *data transfer iterations*. The duration from 0 to $I_p(= T)$ represents a tile of a loop. We define **production** as bringing data from the memory into the cache for a

[1]Speed means coefficient of the iterator.

[2]Distance is defined as the difference of the constant terms when the speeds are the same. With different speeds it is the difference between two access functions.

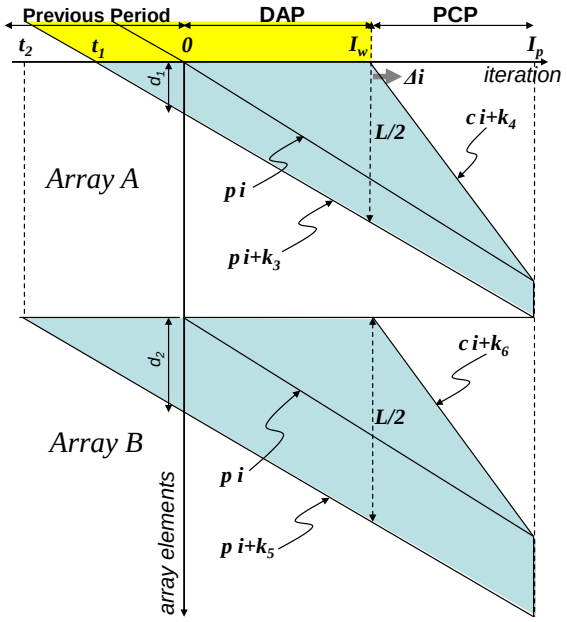


Figure 3: Array-iteration diagram for a type 4 example.

speculative use, performed by the prefetch engine, and **consumption** as the use of data by the processor with no expected reuse of the data in the near future. There are two sets of lines in the diagram. The ones with slope p are production lines, for data prefetch. The ones with slope c are consumption lines, representing the iterations at which each line of array elements is last used by the processor. Thus a line of array elements is present in the cache from the production line until the consumption line. The area bounded by the two lines (shaded area in the figure) represents the useful cache lines at each iteration, and the largest height of the area at I_w signifies that the number of useful cache lines is the maximum at I_w. I_w is the data transfer iteration of the moment when the processor is woken up, and divides a tile into two phases: DAP (Data Accumulation Phase) and PCP (Production-Consumption Phase).

The ratio (γ) between production rate and consumption rate is constant for every reference in a loop, and is determined by the ratio of I_w and I_p. Since the amount of production during I_p should be equal to the amount of consumption during $(I_p - I_w)$, there is such a relationship: $c_i/p_i = I_p/(I_p - I_w) = \gamma > 1$.

Optimal PICA prefetching schemes should meet the following constraints. **(Constraint 1)** In order to maximize the processor idle time stretch for the given data production and consumption rates, we need to make the most use of the cache. This means that the number of useful cache lines at I_w should be as close to the cache size as possible. **(Constraint 2)** At the same time we have to make sure that the cache evicts only those cache lines that have become useless (or have been consumed by the processor). We refer to this constraint as *no eviction of useful cache lines*. Since our assumption is that the cache, whenever necessary, blindly evicts the oldest lines, the utilization of the cache may have to be sacrificed to some degree to satisfy this more important constraint.

4.4 Solutions

In a type 4 loop all the references have the same speed. Figure 3 illustrates the array-iteration diagram. There are two arrays and two references accessing each array all at the same speed; all the production lines have the same slope (p) and all the consumption lines have the same slope (c). The distance between references of an array is denoted as d_i, and d_2 is greater than d_1 as indicated in the figure.

At I_w the number of useful lines is the maximum: $d_1 + d_2 + 2p\,I_w$. For the maximal use of the cache we would like this value to be equal to the total number of cache lines; however, this cannot be achieved unless $d_1 = d_2$. The problem occurs in the DAP phase. In the DAP phase the question is whether all the d_i lines reused from the previous tile can remain in the cache until I_w. Let t_1 be $(-d_1/p)$ and t_2 be $(-d_2/p)$. All the useful cache lines at I_w have been brought in between t_2 and I_w iterations, and can be classified as follows, where negative iterations mean iterations into the previous tile.

- During $t2 \sim t1$: $p(t_1 - t_2) = d_2 - d_1$ lines brought from array B
- During $t1 \sim 0$: $p(-t_1) = d_1$ lines each from arrays A and B
- During $0 \sim I_w$: $p\,I_w$ lines each from arrays A and B

It is obvious that all the lines brought after t_1 will remain in the cache until I_w, since there are only $2d_1 + 2p\,I_w$ lines that are read after t_1, which is less than $d_1 + d_2 + 2p\,I_w$. However, in order to make the $(d_2 - d_1)$ lines from array B read between t_2 and t_1 remain in the cache until I_w, we have to make the same number of lines from array A remain in the cache as well (the area filled in yellow)—there is no distinction between the two arrays from the cache's perspective. This is why we cannot fully utilize the cache at I_w; we have to keep $(d_2 - d_1)$ useless lines from array A as well. Thus the optimal value of I_w is given as $I_w = (L/N - \max_i(d_i))/p = L/Np - \max_i(d_i/p)$, where L is the number of cache lines and N is the number of arrays in the loop.

To guarantee no eviction of useful cache lines in the PCP phase, consider the Δi iterations after I_w. During this time, $2p\Delta i$ new lines are brought into the cache, causing the same number of lines to be evicted. Since in the steady state the two arrays are symmetrical in that the cache holds the exactly same number of lines from each array and they were brought in during the same time period. Therefore the cache will evict the oldest $p\Delta i$ lines from each array. Since there are at least $c\Delta i$ lines that have become useless by then, all the evicted lines are indeed useless lines. This proves that there is no eviction of useful cache lines, and therefore PICA can be successfully applied to type 4 loops.

Type 5 is a combination of type 2 and type 4. While not shown here, it is possible to show that the optimal I_w parameter for type 5 is $I_w = L/\sum_i p_i - \max_i(d_i/p_i)$, which is also the general formula for all the five types.

4.5 Memory Speed

Suppose that a loop has N references to prefetch and let each reference have production rate p_i, which can be found analytically from the optimized code. Then this loop makes on average $\sum_i p_i$ line requests every iteration. Using cycle-per-line measure (*CPL*), which is the average number of cycles that it takes to bring in one line of data into the cache, the number of data transfer cycles per iteration (D) is $D = CPL \sum_i p_i = W_{line} \cdot r_{clk}/(W_{bus}\,\alpha) \sum_i p_i$, where W_{line} and W_{bus} are the widths of the cache line and the bus, respectively, r_{clk} is the ratio of clocks between the bus and the processor core, and α is a positive number between 0 and 1 representing the memory efficiency. The number of computation cycles per iteration (C) can be found easily through simulation for a perfect cache, or $C = N_{instr} \cdot CPI$, where N_{instr} is the number of instructions in the loop body and *CPI* is the cycle-per-instruction of the processor. Then the relationship between I_p and I_w is as

Table 3: Calculation of optimal parameters

Type	$p_i; d_i$	$\sum p_i$	D	C	γ	I_w	w	T
1	1/2, 1/2, 1/2	1.5	48	9	5.3	150	225	184
2	1/4, 1/2, 1	1.75	56	8	7	128	225	150
3	1; 64	1	32	8	4	217	217	289
4	1, 1; 32, 64	2	64	13	4.9	104	209	131
5	1/2, 1; 32, 64	1.5	48	14	3.4	142	213	200

follows.

$$\gamma = I_p/(I_p - I_w) = D/C = CPL\sum_i p_i/C$$
$$= W_{line}/W_{bus} \cdot r_{clk}/(\alpha \cdot C)\sum_i p_i$$

Memory-boundness means that this ratio should be greater than 1. Once the optimal value of I_w is determined, the two PICA parameters w and T can be determined easily: $w = I_w \sum_i p_i$, $T = I_p = I_w \cdot \gamma/(\gamma - 1)$.

5. EXPERIMENTS

In our experiments we use our XScale processor simulator, which has been validated against the 80200 XScale Evaluation Board [7] to be accurate within 7% on average in cycle count measurement. The prefetch engine is modeled in the simulator. We compile both the original and the transformed code using the GCC compiler targeted for the XScale architecture with -O3 option.

XScale L1 data cache is configured to be 32-way 32K bytes with 32-byte lines unless noted otherwise. We assume write-back, first-in-first-out policy. We assume the processor-memory clock ratio of 8.[3] We use the energy models from previously published literature [3, 14]. Our XScale processor runs at 600MHz and can be either Run state (450 mW active, 112.5 mW stall) or MyIdle state (50 mW), the latter of which is an extension of the IDLE state with prefetch, during which the data cache and the prefetch engine are kept active. Every memory access (8-word transaction) consumes additional energy of $(9.46 + 32.5)$ nJ. Only dynamic power is considered in our experiments; however, including leakage power will lead to greater power savings by our methods.

5.1 Validation of Analytical Model

We perform the validation comparing analysis-predicted results against exploration-found results. Based on the problem classification we generate a loop for each of the first five types. References included in those loops are read-only. Table 3 lists the main parameters of the five loops and their predicted optimal values of w and T using our analytical model. Then we also perform extensive parameter exploration on T using simulation. The reason why we chose to perform parameter exploration only on T and not on w is that in many cases the optimal values of w can be found trivially—the number of available cache lines minus, if any, the number of reused lines—and does not vary much depending on the application (see Table 3). For the validation experiments we use a smaller cache with only 256 lines to reduce the search space for exploration. Out of the 256 lines we assume that only 225 lines are available to the PICA technique. But since we use the same values of w both in the analysis and in the exploration, it does not invalidate our experiments. To see the stead state effect we also vary the number of iterations from 1024 to 7168.

[3]Although 8 times slower bus and memory clock may seem too slow for the processor running at 600 MHz, many Systems-on-Chips have several IP cores heavily accessing the external memory, reducing the effective bandwidth available to the processor and increasing the effective clock ratio.

Table 4: Exploration vs. analysis results

Type	T^*	T	E^*	E	E_0	$(E - E^*)/E^*$
1	175	184	0.577	0.596	1.106	3.3%
2	150	150	0.674	0.674	1.312	0.0%
3	250	289	0.383	0.399	0.884	4.2%
4	125	131	0.786	0.813	1.470	3.4%
5	175	200	0.594	0.623	1.121	4.9%

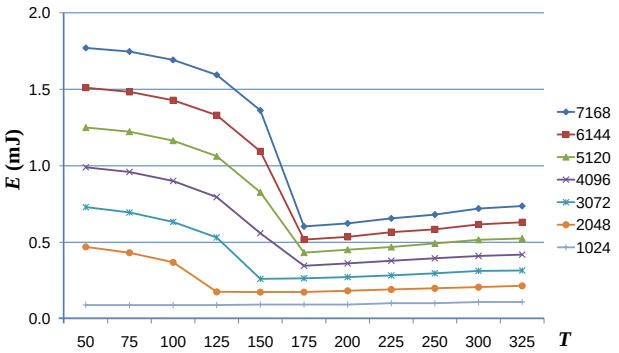


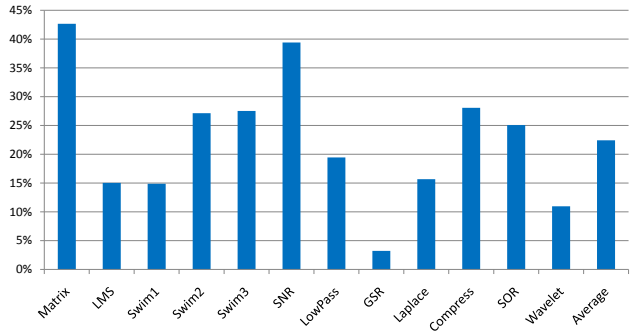
Figure 4: Energy for different T (type 5).

Figure 4 plots the system energy consumption (in mJ), which is the sum of processor energy and bus and memory energy, for different values of T and iteration count (N) for type 5. Other results are similar to the type 5 results, and runtime results are also similar to energy results. Table 4 compares the exploration-found results with analysis-predicted values for all the five cases. Through exploration we first find the optimal parameters (T^*) and their system energy (E^*). Then the analytically computed parameters (T) are used in a simulation to find the system energy for those parameters (E). E_0 is the system energy without PICA. These comparisons are made for $N = 7168$. In all these cases we observe that in the steady state analytically found values are relatively close to experimentally found values (the largest difference is 39 for type 3). More importantly, the difference in the energy is very small, all less than 5%, and far less than the differences from E_0. This demonstrates that our analytical model can predict with accuracy the steady-state optimal PICA parameters for the five classes of loops. For more complicated loops (such as those with conditionals inside the loop body) we can employ simulation-based exploration, even when our analytical model can be used to determine the starting point as well as to reduce the search space.

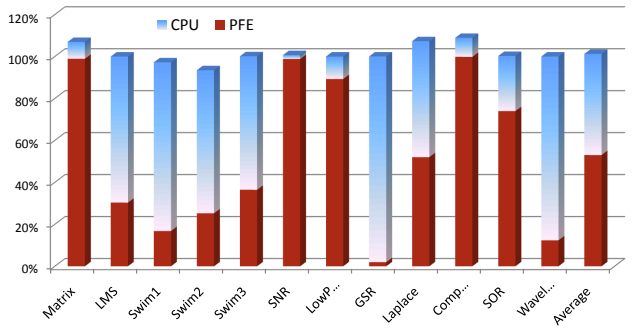
5.2 Benchmark Results

To demonstrate the usability of our enhanced PICA technique we apply PICA to memory-bound kernels in various benchmarks. We use kernels from DSPstone [15], SPEC 95 [16], and multimedia applications. Unlike in our earlier work [3], thanks to the deadlock-free mechanism we can apply PICA to any memory-bound loops, creating a larger set of kernels available for our experiments. Also, all the kernels we use fall into the first 5 types: type 1 (Swim3, SNR), type 2 (Matrix), type 3 (LowPass, Compress, Wavelet), type 4 (LMS, Swim1, Swim2, Laplace, SOR), and type 5 (GSR). Memory accesses that are hard to prefetch, such as writes, irregular memory accesses, and arrays that are already in the cache, are ignored for the classification.[4] For each kernel we optimize the

[4]The reported system energy does include all the memory accesses (e.g., irregular memory accesses, writes). Also, the ratio between



(a) System energy reduction



(b) Memory access profile

Figure 5: Kernel simulation results.

PICA parameters for system energy, using both analytic method and exploration-based fine-tuning together. It took about 1 $\sim$ 2 hours of manual work to analyze each application and instrument the code for PICA optimization, and the exploration-based fine-tuning took typically less than one hour, thanks to the reduced search space by our analytical method.

Figure 5 (a) shows the system energy reduction by our PICA technique over when PICA is not used. The experimental results indicate that (1) our PICA technique is applicable to many memory-bound loops with consistent system energy improvement, (2) when the parameters are fully optimized, using exploration our PICA technique can reduce the system energy up to 42% compared to without PICA.

While energy improvement depends on a number of factors including γ ($= D/C$), cache size, and iteration count, Matrix kernel suggests that γ is very important. In Matrix, where two 2D arrays are multiplied, one array has to be accessed on the higher dimension. This greatly increases D since for every iteration at least one line of cache has to be fetched from memory, which is very rare in other applications. Thus if these memory accesses can be delegated to the prefetch engine, CPU will be relieved of much of its work and the processor energy, as well as the system energy, can be reduced considerably.

Figure 5 (b) plots the memory access profile with our PICA technique, i.e., how many accesses are generated by the prefetch engine (PFE) vs. by the CPU. The scale is normalized to the number of memory accesses generated without PICA (measured in transactions). Thus in the graph every total being near 100% means that the total number of memory accesses generated is roughly the same irrespective of PICA. With PICA, however, the more memory accesses the PFE does, the larger the energy saving will be. Indeed we see a strong correlation between energy reduction in (a) and the PFE portion of memory access profile in (b).

prefetch engine-issued accesses vs. CPU-issued (e.g., irregular) ones is shown in Figure 5 (b).

6. CONCLUSION

Our enhanced PICA greatly improves robustness and applicability of processor stall aggregation based power reduction techniques. First our enhanced PICA can guarantees functional correctness for any set of parameters. Second our enhanced PICA facilitates exploration-based parameter optimization, with which we can fine tune optimal PICA parameters for any memory-bound loop, thus greatly improving applicability of the technique. Finally, since exploration based PICA parameter optimization may be time consuming, we developed static analysis for most common types of loops, thus further improving on its applicability.

Being based on large scale prefetching, our work can provide the basis to study the effect of cache pollution in large scale prefetching, which we intend to further optimize. We also plan to extend our analytical framework to support multi-nested loops and more complex programs.

7. REFERENCES

[1] K. Choi *et al.* Fine-grained dynamic voltage and frequency scaling for precise energy and performance tradeoff based on the ratio of off-chip access to on-chip computation times. *IEEE Trans. CAD*, 24(1):18–28, 2005.

[2] M. Gowan *et al.* Power considerations in the design of the alpha 21264 microprocessor. In *DAC*, pages 726–731, 1998.

[3] A. Shrivastava *et al.* Aggregating processor free time for energy reduction. In *CODES+ISSS*, pages 154–159, 2005.

[4] A. Azevedo *et al.* Profile-based dynamic voltage scheduling using program checkpoints. In *DATE*, page 168, 2002.

[5] T. Burd *et al.* Design issues for dynamic voltage scaling. In *ISLPED*, pages 9–14. ACM, 2000.

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

[7] Intel Corporation. *Intel 80200 Processor based on Intel XScale Microarchitecture*. [Online]. Available: http://www.intel.com/design/iio/manuals/273411.htm.

[8] J. Rabaey and M. Pedram, editors. *Low Power Design Methodologies*. Kluwer Academic Publishers, 1996.

[9] T. Mowry *et al.* Design and evaluation of a compiler algorithm for prefetching. In *ASPLOS*, pages 62–73, 1992.

[10] S. VanderWiel *et al.* Data prefetch mechanisms. *ACM Computing Surveys*, 32(2):174–199, 2000.

[11] E. Brockmeyer *et al.* Layer assignment techniques for low energy in multi-layered memory organisations. In *DATE*, pages 1070–1075, 2003.

[12] I. Issenin *et al.* Data reuse analysis technique for software-controlled memory hierarchies. In *DATE*, pages 202–207, 2004.

[13] M. Kandemir *et al.* Compiler-directed scratch pad memory hierarchy design and management. In *DAC*, pages 690–695, 2002.

[14] A. Shrivastava *et al.* Compilation techniques for energy reduction in horizontally partitioned cache architectures. In *CASES*, pages 90–96, 2005.

[15] V. Zivojnovic *et al.* DSPstone: A DSP-oriented benchmarking methodology. In *ICSPAT*, 1994.

[16] SPEC 95. *Standard Performance Evaluation Corporation*. http://www.spec.org/.