A

PICA: Processor Idle Cycle Aggregation for Energy Efficient Embedded Systems

JONGEUN LEE, Ulsan National Institute of Science and Technology
AVIRAL SHRIVASTAVA, Arizona State University

Processor Idle Cycle Aggregation (PICA) is a promising approach for low power execution of processors, in which small memory stalls are aggregated to create large ones, enabling profitable switch of the processor into low-power mode. We extend the previous approach in three dimensions. First we develop static analysis for the PICA technique and present optimal parameters for five common types of loops based on steady-state analysis. Second, to remedy the weakness of software-only control in varying environment, we enhance PICA with minimal hardware extension that ensures correct execution for any loops and parameters and thus greatly facilitates exploration-based parameter tuning. Third, we demonstrate that our PICA technique can be applied to certain types of nested loops with variable bounds, thus enhancing the applicability of PICA. We validate our analytical model 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*

General Terms: Algorithm, Design

Additional Key Words and Phrases: Low power, code transformation, embedded systems, memory bound loops, processor free time, stall cycle aggregation

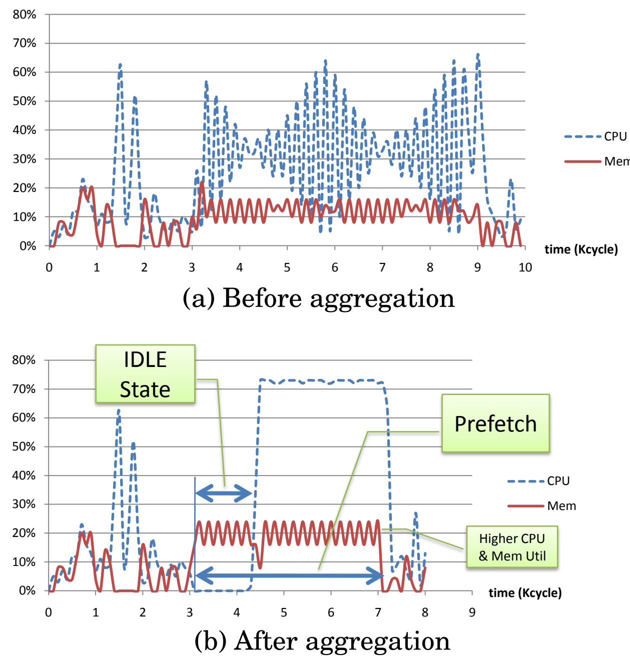
ACM Reference Format:
Lee, J. and Shrivastava, A. 2011. PICA: Processor idle cycle aggregation for energy efficient embedded systems. ACM Trans. Embedd. Comput. Syst. V, N, Article A (January YYYY), 27 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

Energy consumption 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 energy consumption of embedded processors is an important research concern.

Many low power design techniques fundamentally save power when the full performance is not needed. While Dynamic Voltage and Frequency Scaling (DVFS) techniques [Choi et al. 2005] attempt to discover execution intervals when the processor can be slowed down, Dynamic Power Management (DPM) implemented using clock gating, power gating, etc. [Gowan et al. 1998] 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; for instance, the Intel XScale processor

Author's addresses: Jongeun Lee, School of ECE, Ulsan National Institute of Science and Technology, Ulsan, Korea; Aviral Shrivastava, Department of CSE, Arizona State University.
Permission to make digital or hard copies of part or all 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 show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© YYYY ACM 1539-9087/YYYY/01-ARTA $10.00
DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000



(a) Before aggregation

(b) After aggregation

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

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, our cycle-accurate simulation result reveals that when the Intel XScale is executing, practically no stall is more than 360 processor cycles, even though the processor may experience memory stalls almost 30% of the time for a particular application. 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.

Alternatively, the processor stall cycle aggregation approach [Shrivastava et al. 2005] collects several small stalls together to create a large stall in memory-bound loops. Processor power can be saved during the large stall by switching the processor to low-power mode. 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.

To illustrate the aggregation idea, Fig. 1 shows the computation and data transfer rates for a simple loop, before and after applying processor aggregation. The curves are obtained from cycle-accurate simulation using our simulator[1] and represent the number of useful cycles per every 100 cycles, or in other words, CPU and memory bus utilization. The two graphs are identical during the first 3 Kcycles, where program ini-

[1] Details of our simulator is described in Section 8.1

tialization is performed. Soon afterwards the main loop begins in both cases, and we observe computation and data rates increasing. However the patterns are quite different. In (a) the computation and data transfer operations are scattered more or less randomly throughout the loop execution that lasts until about 9 Kcycles, but the graph in (b) has two distinct regions, one with computation and one without. The one without computation starts first. In this region the processor is idle (actually in the IDLE state) for about 1 Kcycles, and only data transfer, or prefetch, is performed. The second region begins when the prefetch engine wakes up the processor, which can run now much faster because there is no memory stall. The data prefetch is continued throughout the second region until both computation and data transfer die out at around the same time. We observe that the aggregation case has overall higher resource utilization and shorter runtime, which is in part due to large scale prefetching employed in the aggregation technique.

While processor stall cycle aggregation can reduce processor energy consumption without performance penalty for memory-bound loops, a simple aggregation may face several problems and limitations. In this paper, we identify a set of necessary extensions and analyses to significantly enhance the aggregation technique both in terms of applicability and energy efficiency, and call it PICA (Process Idle Cycle Aggregation).

— Previous aggregation approach [Shrivastava et al. 2005] depended on the compiler to estimate key parameters of aggregation, i.e., wake-up time w, which is the number of lines that the prefetch engine should fetch before waking up the processor. However the wake-up mechanism might not work sometimes due to various causes ranging from non-determinism in the memory to simple misprediction, which results in deadlocks. In this paper we enhance the aggregation mechanism with hardware deadlock prevention. 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* memory-bound[2] loop.
— The 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 loops present, and present code analysis techniques for them. Through the combined use of static analysis and exploration-based fine tuning, PICA achieves 20% energy reduction on average on a variety of memory-bound loops.
— In the previous proposal, PICA can be applied only to simple loops without nesting or variable bounds. In this paper we demonstrate that PICA can be easily extended for certain types of multi-nested loops, and applied even to loops with variable bounds. This makes PICA applicable to a wider class of important applications.

The rest of the paper is organized as follows. In Section 2 we discuss the related work. In Section 3 we present the basic and deadlock-free versions of the PICA technique. In Section 4 we present the microarchitecture for PICA. In Section 5 we present the static analysis framework for PICA optimization and Section 6 provides analytic solutions to several types of loops. In Section 7 we extend PICA for variable bounds and nested loops. In Section 8 we present our experimental results and conclude the paper in Section 9.

2. RELATED WORK

Perhaps the most popular approach to reducing processor energy is DVFS [Azevedo et al. 2002; Choi et al. 2005]. Especially, [Choi et al. 2005] has some similarity to our work in that it also exploits memory stall cycles by lowering the CPU frequency

[2]For computation-bound loops, prefetching may be more appropriate.

during regions with frequent memory stalls. However, DVFS 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 [Burd and Brodersen 2000]. This and more considerations result in high transition overhead for DVFS. For instance, the Intel XScale processor has frequency switching time of $20\mu s$ or 18,000 processor cycles on a 600 MHz processor [Intel Corporation b]. Consequently, in order to hide the penalty of voltage regulation, DVFS is applied at the context switching granularity (tens of milliseconds), typically by the operating system. Our PICA approach exploits IDLE mode, whose transition time is merely 180 processor cycles.

In contrast, DPM implemented using clock gating, power gating, body biasing, etc. [Gowan et al. 1998; Rabaey and Pedram 1996; Benini et al. 2000] 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. For instance, fetch throttling [Unsal et al. 2002] throttles the fetch unit in order to reduce the processor power based on the compiler-generated IPC (Instruction Per Cycle) information. Similarly, [Gowan et al. 1998] reduces processor energy by controlling a small part of the processor such as multiplier and floating point unit; the inactive parts of processor are statically or dynamically estimated, and switched to a low-power mode. 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. Prefetching [Mowry et al. 1992; VanderWiel and Lilja 2000] has been studied as a means to reduce memory latency in applications exhibiting low-locality access patterns such as scientific and data-intensive applications. Reducing memory latency has many benefits even including reducing the energy consumption of an application. An extensive survey [VanderWiel and Lilja 2000] on data prefetching mechanisms classifies data prefetching into software and hardware techniques. Software prefetching relies on *fetch* instructions inserted in the instruction stream, where each fetch instruction can load one or only a small number of cache lines from the lower memory. Hardware prefetching relies on speculation and thus is more likely to generate unnecessary memory traffic and increase cache pollution, but it can eliminate the overhead of fetch instructions and utilize runtime information. All these techniques fall in the class of small-scale prefetching, which is widely used and an accepted form of prefetching nowadays.

However, 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. Though scratch pad management techniques [Brockmeyer et al. 2003; Issenin et al. 2004; Kandemir and Choudhary 2002] 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."

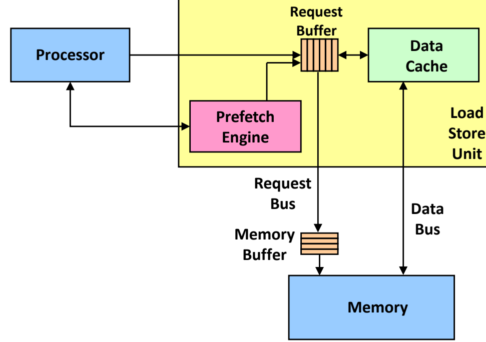
While there are elaborate analytical models on cache behavior [Ghosh et al. 1997; Chatterjee et al. 2001; Verdoolaege et al. 2007; Shrivastava et al. 2010], none of them are widely used today, as they have practical limitations such as high computational complexity and limited architectural features supported. Moreover, our problem is more complex than cache modeling, because it also involves hardware prefetching and multiple processor states. Our approach is to use simple analytical models first to quickly find the profitability of PICA transformation and then to use simulation-based fine-tuning of PICA parameters.

```
1: for ( i = 0; i < N; i++ ) {
2:   Statements involving i
3: }
```

(a) Original loop

```
1: setPrefetchArray addr, #items,
   . . .
2: startPrefetch
3: for ( j = 0; j < N; j += T ) {
4:   procIdleMode w
5:   M = min(j + T, N)
6:   for ( i = j; i < M; i++ ) {
7:     Same statements
       involving i
8:   }
9: }
```

(b) PICA-transformed loop



(c) Architecture

Fig. 2. PICA transformation and architecture.

Table I. How deadlock can happen

(a) Expected line fetch scenario

Tile number	1	2	3	4
#lines fetched during the idle mode	50	50	50	50
#lines fetched while processor is awake	50	50	50	50

(b) Deadlock scenario

Tile number	1	2	3	4
#lines fetched during the idle mode	50	50	50	40
#lines fetched while processor is awake	70	70	70	–

3. PICA APPROACH

PICA is a compiler-microarchitecture cooperative technique exploiting large-scale prefetching for a very efficient loop execution. Here we present the code transformation and microarchitecture for PICA. First we describe the basic mechanism and then present extensions to enhance the applicability and effectiveness.

3.1. Basic Mechanism

Figures 2(a)-(b) illustrate the loop transformation that is required to perform PICA. In addition to setting up the prefetch engine for the required data before the loop begins (lines 1–2 in (b)), the original loop, that ran from $i = 0$ to $i = N - 1$, has to be tiled into "tiles" of size T (lines 3 and 5–6). 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 works on the data present in the cache. But since it *consumes* data faster than memory can *produce*, eventually the prefetched data in the cache will be used up. T is determined such that the tile ends just when all the prefetched data in the cache is used up and the processor misses in the cache if it continues anymore. w is determined under the constraint that the prefetch should not overwrite the still-to-be-used data in the cache.

Table II. PICA instructions

Instruction	Description
`setPrefetchArray`	Add to the prefetch list a new prefetch task (see Section 4.2 for detail). Add to $counter2$ the number of additional lines to request.
`startPrefetch`	Start prefetching and start $counter2$ (decrement it by one for each line fetched).
`procIdleMode` w	Set $counter1$ to w and put processor into idle mode only if $w \leq counter2$, otherwise do nothing. Also start $counter1$ (decrement it by one for each line fetched; upon reaching zero wake up the processor).

3.2. Enhancement for Deadlock Avoidance

The main task of PICA transformation is to determine the optimal values of parameters w and T. Once these parameters are fixed, the remaining code transformation is straightforward. However, even after correctly setting up the parameters, the transformed code might not run correctly, but can actually incur a deadlock. Deadlock may happen because the processor wakeup time is controlled only by software.

To illustrate the problem, let us consider a case where the tile size T is set to one fourth of N, the number of iterations of the loop, and w is set to 50 lines out of the total 400 lines to be fetched. It works as follows. During the first tile, the first 100 lines are fetched—50 while the processor is in idle mode and another 50 while the processor is awake. If the computation rate is exactly twice the data fetch rate, this schedule can be the perfect PICA execution. The same pattern is repeated for the remainder of the tiles, and the entire loop is completed in four tiles, as summarized in Table I(a).

This schedule, however, depends too much on the delicate balance between computation and data transfer rates. Suppose that for some reason data transfer rate is 40% higher than expected, as compared to computation rate. This change does not change the second row in Table I(a), but the third row. This is because the processor wakeup time w controls only the number of lines fetched during the *idle* mode, but not while the processor is *awake*. When the computation finally reaches Tile 4, the processor is first put into the idle mode, to be waked up after 50 lines are fetched. However, that never happens because there are not enough lines left to be fetched, resulting in a deadlock situation.

To prevent deadlock, we add to the prefetch engine a simple hardware logic that checks whether there are enough lines to fetch before putting the processor to the idle mode. This should work because the reason for deadlock is simply that there are less than w lines left to be fetched when the processor goes into the idle mode, which effectively invalidates the wake-up parameter. Implementation of this logic requires only one counter for the number of remaining line requests and one comparator.

Now that the prefetch engine knows how many more lines to fetch, it can prevent the processor from going into the idle mode if the number of remaining lines is less than w. This enhancement not only makes the PICA technique more robust but also greatly improves its applicability and effectiveness. In the basic PICA it was the sole responsibility of the compiler to estimate w and T correctly so as to avoid deadlocks, and pessimistic estimates by the compiler to guarantee correctness may result in lower energy reductions. In the enhanced PICA, since the deadlock-free operation is guaranteed, we can aggressively explore the parameter space to find the optimal PICA parameters w and T for *any* memory-bound loop and thus maximize the energy reduction. Further, this extension greatly enhances the applicability of PICA on complex programs, which may be out of the reach of traditional compiler analysis.

4. ARCHITECTURAL REQUIREMENTS

4.1. Prefetch Engine

Figure 2(c) illustrates the processor-memory subsystem architecture including the prefetch engine though the exact implementation of the prefetch engine as well as its interface may differ for each architecture. The prefetch engine is a small piece of hardware inside the load store unit of a processor pipeline, and is controlled by the PICA instructions as described in Table II. The prefetch engine mainly deals with the prefetching of array data from the memory to the data cache, but also handles PICA-related processor controls such as putting the processor into the low power mode and waking it up.

To explain the behavior of the three PICA instructions let us consider the example in Fig. 2(b). First, the `setPrefetchArray` instruction programs the prefetch engine to generate a set of line requests upon initiation. More than one `setPrefetchArray` instructions can be used at the same time, and their parameters such as the start address, the stride, the number of items, etc. follow directly from the reference expressions of the original code. Second, the `startPrefetch` instruction simply initiates the prefetch engine, which then keeps prefetching by inserting line requests in the request buffer. The prefetch engine monitors the request buffer to ensure that the request buffer is full as long as possible. While the request buffer is full, the activity on the data bus remains uninterrupted, ensuring maximal data bus usage.[3] Third, on entering a new tile the processor is put into the low power mode by executing the `procIdleMode` instruction. In the low power mode the clock to the processor, except the load store unit, can be frozen, and even powered down. The `procIdleMode` instruction takes one parameter, which is the wake up parameter w, or the number of lines to fetch before waking it up. As soon as the prefetch engine fetches w lines from the memory, the processor is woken up either with interrupt or by other means. The processor then resumes the execution through the rest of the tile. The prefetch engine continues its operation even after waking up the processor. After all the data has been fetched (which should ideally coincide with the end of the loop), the prefetch engine should disengage itself until it is re-invoked by the processor for the next loop.

The prefetch engine maintains two counters. The first one, $counter1$, is used to track the processor idle period. It is set by a `procIdleMode` instruction to the wake-up parameter w, and decremented by one for every line fetched. As soon as it reaches zero, the prefetch engine wakes up the processor. The second one, $counter2$, is used to track the number of remaining lines to fetch. It is increased by each `setPrefetchArray` instruction by the number of lines to fetch, and decremented by one, similarly to $counter1$, for every line fetched. Note that the calculation of the number of line requests here does not have to be exact, but only needs to be consistent with decrements. For instance, the prefetch engine does not re-fetch data that is already in the cache, which would be extremely difficult to predict in advance. We resolve this difficulty by decrementing the counters even for a line that is already in the cache and therefore not fetched. Then setting up the $counter2$ parameter for `setPrefetchArray` instruction does not need to account for the lines that may be already in the cache.

4.2. Prefetch Parameters

Since the PICA technique relies on prefetch for most of its memory operations, it is important to clarify the kind of prefetch used in our scheme. While much of the previous work discusses small-scale prefetch, where one or a few line requests are made in

[3]The bus and the memory are assumed to be pipelined with separate request and data channels so as to maximize the throughput. This also makes it easier to find the memory bandwidth in our static analysis.

advance by the processor [VanderWiel and Lilja 2000], we employ large-scale prefetch. The prefetch engine is programmed for an entire loop, and once initiated, it operates autonomously, even controlling the power state of the processor. Moreover there may be multiple arrays of data that need to be prefetched. Thus the challenge is not only to specify the memory access pattern for an entire loop, but also to direct the prefetch to be done in a way that maximizes data reuse in the cache, which is crucial to increase the energy saving by PICA.

Our `setPrefetchArray` instruction has four parameters: start address, stride, the number of items (the unit of prefetch is a cache line), and weight. First note that our PICA technique does not require *every* memory access to be prefetched, since any data that is not found in the cache will trigger a cache miss and be fetched on-demand (called *CPU-generated memory access*), but the greater the number of prefetch-generated memory accesses is, the greater the energy saving also becomes, as is shown in our experiments. We target sequentially accessed memory accesses, which are most commonly generated by array references in a loop (e.g., $A[i]$). The start address and stride can be easily found from array reference expressions. Calculating the number of items requires the length of the loop, or the number of iterations. The number of iterations does not have to be a constant as long as it is fixed and known before the loop entry.

The last parameter, *weight*, is needed to balance the speed of prefetch between multiple array references. For instance, if two memory references $A[i]$ and $B[2i]$ need to be prefetched, the number of lines needed for each will be different, i.e., two lines of $B[2i]$ for every line of $A[i]$. Since we want the two streams of prefetch to end simultaneously, we should speed up prefetching $B[2i]$ twice as fast as $A[i]$. This can be achieved by employing weighted round-robin scheduling in the prefetch scheduler, by doubling the weight for $B[2i]$ than for $A[i]$. In general the weight can be given proportionally to the *reference speed* of a reference, where reference speed is more precisely defined in Section 5.1.

5. 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 the 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.

5.1. Input

Since our PICA transformation as well as the prefetch engine can support only one-dimensional loops, we need to consider only one loop level, which may be any in a loop nest. Accordingly, only one iterator, or loop induction variable, is relevant in our PICA analysis, with all the other iterators regarded as constants. References to multi-dimensional arrays can be easily converted to references to single-dimensional arrays since array dimensions are statically known. While any reference expressions can be supported in our PICA transformation, our analysis is concerned with only those that are affine functions of the iterator. An affine function has two parameters, coefficient and constant, which are used to determine the *speed* and *distance* of references as follows.

The *speed* or 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 array A has 4-byte elements and

Table III. Loop classification

Type	AM	RM	SS	Example
1	Multi	Single	All references	$A[i] + B[i] + C[i]$
2	Multi	Single	None	$A[i] + B[2i]$
3	Single	Multi	All references	$A[i] + A[i + 10]$
4	Multi	Multi	All references	$A[i] + A[i + 10] + B[i] + B[i + 20]$
5	Multi	Multi	Only references accessing the same array	$A[i] + A[i + 10] + B[2i] + B[2i + 30]$
6	Single	Multi	None	$A[i] + A[2i]$
7	Multi	Multi	None	$A[i] + A[2i] + B[i + 10] + B[3i + 15]$

the data cache has 32-byte lines, a reference $A[i + k_1]$ will need a new cache line every eighth ($= 32/4$) iteration, assuming that the iterator i is incremented by one. Therefore the production rate of this reference is $p_1 = 1/8$. Let α denote the ratio between the cache line width and the array element width. In general, the production rate of a reference $A[a\,i{+}k_2]$ is given by $\min(|a|/\alpha, 1)$, assuming that the iterator i is incremented by one. *Distance* is defined between two references, with the same coefficient, to the same array, as $\lceil |k_3 - k_4|/\alpha \rceil$, where k_3 and k_4 are the constants. Distance is defined in terms of cache lines and independent of the coefficients.

In the steady state a write reference is equivalent to two read references with the same speed in terms of data transfer. Thus we consider only read accesses in our analysis. For static analysis we assume fully-associative caches and FIFO (First-In-First-Out) replacement policy.

5.2. Loop Classification

Our assumption of references being affine functions allows us to classify loops into seven types listed in Table III. We also studied several multimedia and DSP applications to find out all the memory-bound loops. Most of the loops with compile-time deterministic access patterns fall into the first five types, for which we analytically compute the optimal PICA parameters w and T in Section 6. The table 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 are required to have the same coefficient and thus the same speed (*SS*).

Type 1 is the simplest and a trivial generalization of the only case addressed in [Shrivastava et al. 2005]. We earlier presented our analysis for type 4 loops in [Lee and Shrivastava 2008] but this paper presents the analysis for all five types of loops. Types 3 and 6 are special cases of types 4 and 7, respectively, and can be easier to handle. Types 2 and 5 allow different arrays to be accessed at different speeds whereas the last two types allow even references to the same array to have different speeds. In the latter case, accurate steady-state analysis becomes very difficult, since the distance between references with different speeds is not well-defined and can be time-varying. In fact there are two modes—when the distance is short enough the references can be considered to be *overlapping*, but when the distance is long they behave just like references to different arrays. Because the distance changes over time, we cannot determine the parameter value that is optimal throughout the entire iterations.

5.3. 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 Fig. 3. The vertical axis represents the array elements in terms of cache lines. The horizontal axis represents time in terms of *data transfer iteration*, where is the iteration of which the prefetch engine brings the data. We use *data transfer* iteration instead of *computation* iteration, since the latter has no one-to-one correspondence with time due to processor idle periods. The duration from

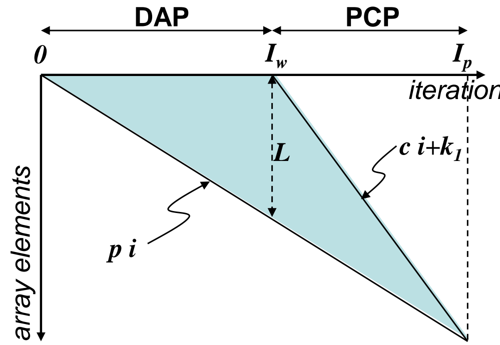


Fig. 3. Array-iteration diagram for a single reference.

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 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 lines in the diagram. The one with slope p is a production line, for data prefetch. The one with slope c is a consumption line, mapping each array element to the iteration at which the array element is last used by the processor. Thus an array element is present in the cache and useful from the production line until the consumption line. The area bounded by the two lines (shaded area in the figure) represents the number of useful cache lines at each iteration, and its height being maximum at I_w signifies that the utilization of the cache is maximized 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 the production rate and the consumption rate is the same for all the references 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 $(I_p - I_w)$, the following relationship holds: $\gamma = c_i/p_i = I_p/(I_p - I_w) > 1$.

Finding the optimal PICA parameters can be viewed as an optimization problem with an objective and a constraint. **(Objective)** 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)** At the same time we have to make sure that the cache evicts only those lines that have become useless, or have been consumed by the processor, since otherwise there will be unnecessary memory accesses, resulting in a significant increase in both runtime and energy. We refer to the 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 the constraint. In the next section we derive the PICA parameters that maximize the objective while meeting the constraint.

5.4. Memory Speed

Suppose that a loop has N references to prefetch and each reference has production rate p_i, which can be easily found by program analysis. 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

```
for ( i = 0; i < 1000; i++ ) {
   A[i+32] = A[i+16] + A[i]
   B[i+512] = B[i]
}
```

Fig. 4. A simple loop.

number of data transfer cycles per iteration (D) can be represented[4] as $D = \textit{CPL} \sum_i p_i$. The number of computation cycles per iteration (C) can be easily found[5] through simulation for a perfect cache. Then the relationship between I_p and I_w is as follows.

$$\gamma \;=\; I_p/(I_p - I_w) = D/C = \textit{CPL} \sum_i p_i/C$$

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 easily determined: $w = I_w \sum_i p_i$ and $T = I_p = I_w \cdot \gamma/(\gamma - 1)$. Hence, our solutions in Section 6 are given only in terms of I_w.

5.5. Overlapping vs. Separate References

Consider the example loop in Fig. 4 and assume that the cache has 64 lines and $\alpha = 8$. Also assume that $I_w = 240$; that is, the processor is woken up after the data transfer finishes prefetching data for 240 iterations. In this scenario, the lines that are accessed in the first I_w computation iterations of each tile by the references of array A are overlapping while those accessed by the references of array B are not.[6] We call same-speed references to an array *overlapping* if the lines accessed by them in the first I_w computation iterations of each tile are overlapping. The other case is called *separate*. If two or more references to an array are separate, they can be considered as references to *different* arrays for the purpose of our analysis, since there is no reuse between them. Hence our static analysis needs to show how to handle same-speed references that are overlapping; the other case, viz., separate references, is trivial.

This definition of *overlapping* references has one problem that it requires I_w to be fixed first. Once we know I_w, it is very easy to check the correctness of the decision, but without the correct decision we cannot arrive at the optimal value of I_w. The problem of determining whether two or more references to the same array are overlapping can be solved as follows. Consider this example: $A[a_1\, i] + A[a_1\, i + c_1] + B[a_2\, i] + B[a_2\, i + c_2]$, where i is the iterator incremented by one and $\alpha = 8$. The distance d_A between the two references to A is $\lceil c_1/8 \rceil$, and similarly $d_B = \lceil c_2/8 \rceil$ for array B. Let the production rates for A and B be p_A and p_B, respectively. Ultimately we want to estimate the number of lines that are accessed by each reference. Let L_A and L_B be the number of lines used by references to A and references to B, respectively, and L be the total number of cache lines. In the steady state, $L_i \propto p_i$. Therefore if all the references are separate, it should be that $L_i^s = L\, p_i/(2p_A + 2p_B)$ and $L_i^s \leq d_i$. If all the references are overlapping, it should be that $L_i^o = (L - d_A - d_B)\, p_i/(p_A + p_B)$ and $L_i^o \geq d_i$. Similar inequalities can be made for mixed cases. For example, if only references to A are

[4] D can also be represented using architectural parameters, as $D = W_{line} \cdot r_{clk}/(W_{bus}\, U) \sum_i p_i$, 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 U is a positive number between 0 and 1 representing the memory bus utilization.

[5] Alternatively, C can be computed as $C = N_{instr} \cdot \textit{CPI}$, where N_{instr} is the number of instructions in the loop body and *CPI* is the cycle-per-instruction, for a perfect cache, of the processor.

[6] During the first I_w iterations the two references of B access $B[512] \sim B[751]$ vs. $B[0] \sim B[239]$, which do not overlap even at the cache line size granularity.

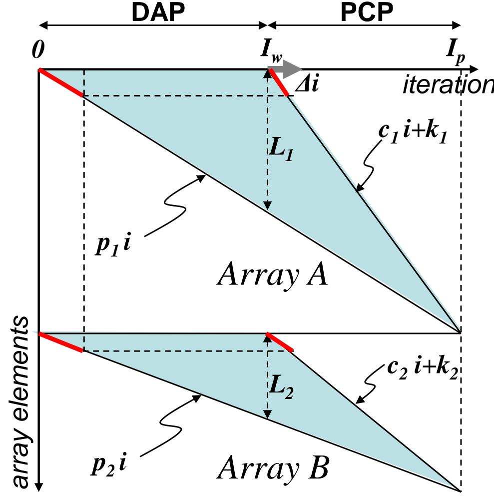


Fig. 5. Array-iteration diagram for type 2: two references with different speeds.

separate, it should be that $L_i^m = (L - d_B)\, p_i/(2p_A + p_B)$ and $L_A^m \leq d_A$ and $L_B^m \geq d_B$. Note that not all those cases can be true and all but one will be self-contradicting. Thus the one that is not self-contradicting must be the correct one. In case there are more than one cases that are not self-contradicting, we take any randomly. This exhaustive search to find out whether some references are overlapping has an exponential time complexity, but only in the number of same-speed references minus the number of unique arrays accessed by them, which is typically very small. Therefore we do not attempt to improve this further.

Array a is accessed by three references, whose access patterns differ only in the constant part. There may exist overlaps between the accesses of the three references. In the case of array a, for instance, the lines requested for one reference may be partially reused by another reference that overlaps with the first one. In contrast, array b is accessed by two references, whose access expressions differ by a large constant. In this case there might not be any overlap within a tile between the accesses unless the cache is very large. Depending on the value of the constant offset, together with other parameters such as cache size, the amount of overlap may vary, which changes the optimal values of the tile size and the wait time. To have a more detailed look at the array prefetch and use together with the iterations and tiles, we introduce *array-iteration diagram*.

6. ANALYTICAL SOLUTIONS

We now present our analytical solutions for the first five types of loops. Type 5, which is the most general of the five types, has two components that make the analysis difficult: different speed and line reuse. We discuss different speed in type 2 (which also covers type 1), and line reuse is discussed in types 3 and 4. Combining types 2 and 4, we can easily derive the optimal solution for type 5.

6.1. Different Speed: Type 2

Example 6.1. $A[a_1 i] + B[a_2 i]$, where $a_1 \neq a_2$.

Type 2 allows only one reference for each array but each reference may have a different speed. The array-iteration diagram for a type 2 loop is illustrated in Fig. 5, which

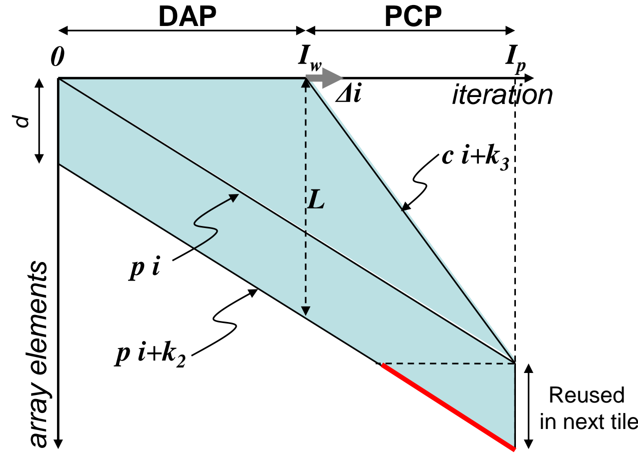


Fig. 6. Array-iteration diagram for type 3: two same-speed references to one array.

has two pairs of production and consumption lines. Production and consumption for one array are independent of those for the other, since we assume that there is no cache conflict (other than capacity misses) or aliasing between different arrays. We approximate the cache behavior by assuming that the number of cache lines used by each reference is proportional to its production rate. Therefore each reference is given the following number of cache lines: $L_i = L\,p_i/\sum_j p_j$, where L is the total number of cache lines and p_i is the production rate of reference i. Then I_w can be easily computed from the diagram as $I_w = L_i/p_i = L/\sum_i p_i$.

Using the array-iteration diagram we can also show that there is no eviction of useful cache lines for type 2. During the DAP phase (data transfer iterations 0 to I_w) the cache will evict only those lines used in the previous tile, since the cache size is enough for the newly fetched lines (shaded area in the figure). Consumption starts at I_w. During the first Δi iterations of a PCP phase, $p_j\Delta i$ lines are brought into the cache for each array (where j is A or B), requiring the same number of lines to be evicted. At the same time $c_j\Delta i$ lines become useless, which are the oldest ones *not only among the lines for the same array but also in the entire cache*. The reason is as follows. From the array-iteration diagram the time when the first $c_j\Delta i$ lines (thick red line segments in the figure) were brought is between 0 and $c_j\Delta i/p_j = \gamma \cdot \Delta i$, which is the same for all the references. Therefore all the $(\sum_j c_j)\Delta i$ lines that are now useless were brought before $\gamma \cdot \Delta i$ and thus are the oldest lines in the entire cache. Since $p_j < c_j$ (i.e., more useless lines than the new lines), it is guaranteed that no useful cache lines are evicted during Δi iterations. Since Δi can be any arbitrary integer between 1 and $(I_p - I_w)$, there is no eviction of useful cache lines in the PCP phase. Type 1 is a special case of type 2 and the same formula applies to type 1 as well.

6.2. Line Reuse with a Single Array: Type 3

Example 6.2. $A[a\,i] + A[a\,i + k_1]$

Figure 6 illustrates the array-iteration diagram with two overlapping references. The cache lines accessed by the leading reference $A[a\,i + k_1]$ are also accessed after some iterations by the trailing reference $A[a\,i]$, assuming $k_1 > 0$. Therefore there are only one production line (the lower one) and only one consumption line for the two references. Let d be the distance between the two references, which is defined in terms

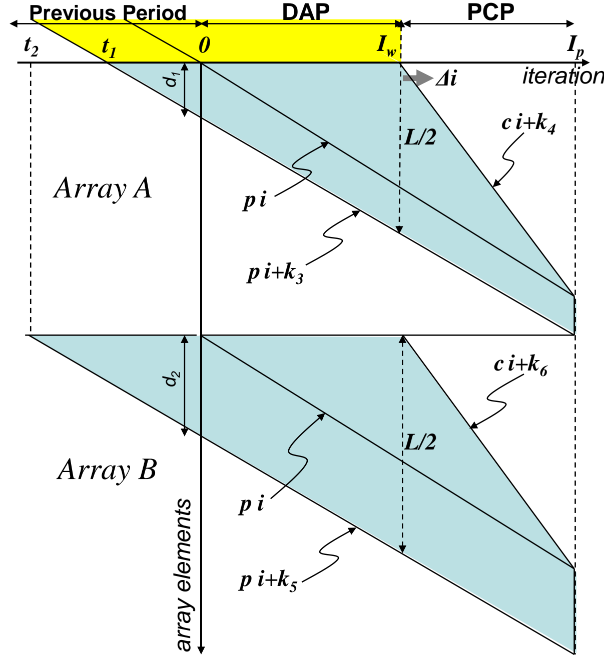


Fig. 7. Array-iteration diagram for type 4: four references accessing two arrays all at the same speed.

of *lines*. At the beginning of DAP there are d useful lines in the cache, which are reused from the previous tile, and production is performed according to the production line. Consumption starts when the number of useful lines reaches L. Therefore from the diagram the optimal value of I_w is given as $I_w = (L - d)/p$.

To show that this value of I_w does not violate the constraint, we consider two cases. (i) PCP phase: During the first Δi iterations of a PCP phase $p\Delta i$ lines are newly fetched, forcing the same number of lines to be evicted. At the same time the $c\Delta i$ oldest lines become useless. Since $p < c$, all the lines evicted during the PCP phase are useless. Thus there is no eviction of useful lines in PCP. (ii) DAP phase: We have to guarantee that all the d reused lines are preserved (not evicted) between successive tiles until it reaches I_w. This holds true if and only if the d reused lines are the newest at the beginning of DAP, since by the end of DAP all from the previous tile except d number of lines are evicted. We see from the diagram that the d reused lines are indeed the most recently fetched ones in the previous tile (see the thick red line segment). Thus there is no eviction of useful lines in the DAP phase.

6.3. Line Reuse with Multiple Arrays: Type 4

Example 6.3. $A[a\,i] + A[a\,i + k_1] + B[a\,i] + B[a\,i + k_2]$

Though type 4 is a simple extension of type 3, unlike in type 3, we cannot fully utilize the cache at I_w. Figure 7 illustrates the array-iteration diagram of a type 4 loop, where there are two arrays each accessed by two references. Since all the references have the same speed, the production lines have one slope (p) and the consumption lines have another (c). The distance between the references of an array is denoted by d_i and we can assume without loss of generality that d_2 be 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 done unless $d_1 = d_2$. The problem occurs in the DAP phase. In DAP 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 fetched 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 fetched 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 fetched after t_1 will remain in the cache until I_w, since there are only $2d_1+2p\,I_w$ lines that are fetched 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 fetched 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 see that there is no eviction of useful 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, forcing the same number of lines to be evicted. Since in the steady state the two arrays are symmetrical in the sense that the cache holds exactly the same number of lines from each array, 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. This proves that there is no eviction of useful cache lines in PCP.

6.4. Combining Different Speed with Line Reuse: Type 5

Example 6.4. $A[a_1\,i] + A[a_1\,i + k_1] + B[a_2\,i] + B[a_2\,i + k_2]$

Type 5 is the combination of type 2 and type 4. For the same reasons as in type 4 we cannot fully utilize the cache at I_w for type 5 loops. To see exactly how many useless lines must be added, let us consider two arrays and two references accessing each array, as illustrated in Fig. 8. The production and consumption rates are p_1 and c_1 for array A, and p_2 and c_2 for array B, respectively. The distance between the references to each array is denoted by d_i. Let t_i be $(-d_i/p_i)$ and t_m be the minimum of t_1 and t_2. Then the lines that remain in the cache at I_w are those and only those that were fetched during t_m and I_w. Therefore the number of lines at I_w is $(I_w - t_m)(p_1 + p_2)$, which should be equal to the total number of cache lines. Thus the optimal value of I_w is given as $I_w = L/\sum_i p_i + t_m = L/\sum_i p_i - \max_i(d_i/p_i)$, which is also the general solution for all the five types.

It is not difficult to see that there is no eviction of useful lines in the PCP phase as well. During Δi iterations after I_w there are $(\sum_j p_j)\Delta i$ lines evicted from the entire cache. The oldest $p_j\Delta i$ lines from each array have been brought into cache between t_m and $t_m+\Delta i$ iterations (regardless of the array) and already become useless, since there are at least $c_j\Delta i$ lines from each array that have become useless. Therefore those $p_j\Delta i$ lines will be evicted from each array and there is no eviction of useful lines in PCP.

7. FURTHER ENHANCEMENTS

In this section we extend the PICA technique in two significant ways, which makes PICA applicable to a wider variety of important applications.

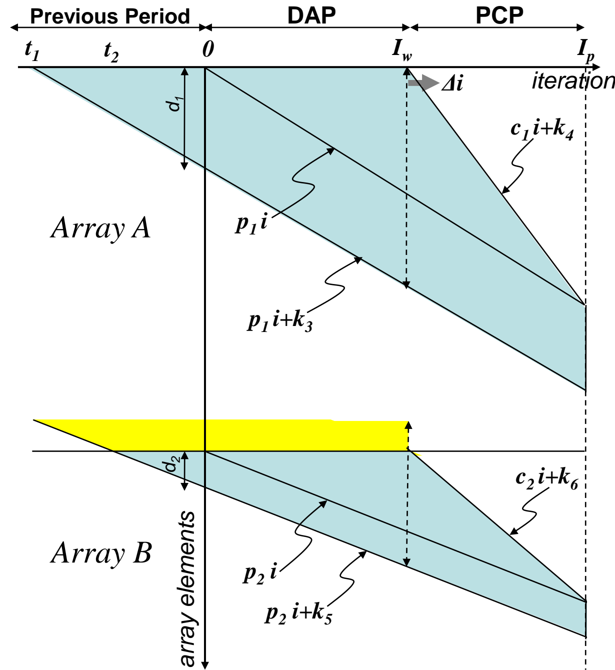


Fig. 8. Array-iteration diagram for type 5: four references accessing two arrays, same speed for same array.

```
1: for ( i = 0; i < X; i++ ) {
2:     A[i] = A[i] + B[i]
3: }
```

Fig. 9. A simple loop with variable bounds, where “X” is a variable.

7.1. Variable Bounds

So far we have assumed that the bounds of a loop are constants, to make it easier to derive our static analysis. However they are often variables in reality, as illustrated in Fig. 9. Here we show that the PICA technique can be applied to certain types of loops with variable bounds.

The variable “X” in our example is needed in two places for PICA transformation: first to set up the prefetch engine, and second to determine the best code transformation for the loop. First, the prefetch engine needs to know exactly how many items of each array to bring. In this example this issue can be resolved easily, since by the time the program reaches the beginning of the loop, the variable “X” should be known, and therefore this value can be used to program the prefetch engine. In general, however, variable bounds may only be fixed later during the loop execution (e.g., “X” is modified in the loop body), in which case the prefetch engine cannot be programmed correctly. Thus we require that the bounds should be known before loop entry.

Second, variable bounds make it more challenging to find code transformations. Note however that in our static analysis the two parameters needed to transform a loop (the tile size and the wake-up parameter) are determined independently of the number of iterations, assuming “X” is large enough. Different values of “X” affect only the number of tiles, and not the tile size itself. This is because the tile size and the wake-up param-

```
1:  for ( k = 1; k < NTIMES; k++ ) {
2:     for ( j = 0; j < 4; j++ ) {
3:        avgtime[j] = avgtime[j] + times[j][k]
4:        mintime[j] = MIN(mintime[j], times[j][k])
5:        maxtime[j] = MAX(maxtime[j], times[j][k])
6:     }
7:  }
```

(a) Original loop copied from the Stream benchmark

```
1:  setPrefetchArray times, #items, ···
2:  startPrefetch
3:  for ( t = 1; t < NTIMES; t += T ) {
4:     procIdleMode w
5:     M = min(t + T, NTIMES)
6:     for ( k = t; k < M; k++ ) {
7:        for ( j = 0; j < 4; j++ ) {
8:           avgtime[j] = avgtime[j] + times[j][k]
9:           mintime[j] = MIN(mintime[j], times[j][k])
10:          maxtime[j] = MAX(maxtime[j], times[j][k])
11:       }
12:    }
13: }
```

(b) PICA can be applied to an outer loop

Fig. 10. PICA transformation for a loop nest.

eter are determined by the memory speed (memory access pattern) and the computation speed, and how long the loop is, is largely irrelevant. But since our scheme does not allow variable parameters (e.g., variable wake-up time), the loop bounds should be constant throughout one loop execution though different executions may assume different sets of bounds.

It is worth mentioning that the same cannot be said about the basic PICA version [Shrivastava et al. 2005]. This is because in the original PICA version, a loop is divided into three sections—prolog, kernel, and epilog—and only the kernel is tiled. This is to avoid a wrong wake-up parameter being used in the epilog, which could cause a deadlock. The new PICA is robust, and even if a wrong wake-up parameter is used in the last tile or epilog, it does not cause a deadlock, and that is why our PICA has only the kernel part and does not need prolog or epilog, and also why we do not really need the number of iterations of a loop in deriving PICA parameters.

In summary, to apply PICA it is required that the loop bounds must be fixed throughout each loop execution and known before loop entry. The actual length of the loop is irrelevant in making PICA transformations, but profitability requires that the loop be long enough.

7.2. PICA for Nested Loops

So far we have considered the PICA transformation only for the innermost loop in a loop nest. This was motivated by the much greater difficulty of large scale prefetching as the number of loop levels increases. For instance, to prefetch an array that is accessed through multiple iterators in a loop nest, the prefetch engine should be able to handle complex memory access patterns generated by various expressions involving the multiple iterators. Even if we limit the array index expressions to be linear, it can

still include many different types of access patterns, which means more complicated set up for the prefetch engine and more complex prefetch engine hardware.

Thus if we are constrained to a single loop level, the obvious choice would be the innermost loop. However, it is not the only one, and applying PICA to an outer loop may prove more effective. Consider the loop in Fig. 10(a), which is copied from the Stream benchmark [McCalpin 1995]. The loop nest has two loop levels but the innermost loop has only four iterations, which is simply too short for PICA. Consequently if we look for innermost loops only, we will certainly miss the opportunity to apply PICA to this kernel.

However, if we consider outer loops as well, the kernel turns out to be a perfect example for PICA, as illustrated in Fig. 10(b). The code transformation used for the outer loop in this example is not very different from the one for innermost loops, and indeed they are the same except that the inner loop is treated like a single statement.[7] However, applying PICA to an outer loop requires more caution due to the limitations the prefetch engine may have. In our example, the "times" array is accessed sequentially throughout the loop nest, which makes it trivial to set up the prefetch for it, as is the case with the other three arrays. But in general an array access pattern may be more complex (e.g., trapezoidal), and thus may be difficult or very inefficient to prefetch, reducing the profitability of PICA. In summary, PICA can be applied to any loop level in a loop nest as long as its inner-loops, if any, have constant bounds so that the PICA parameters can be fixed. This is because a loop with constant bounds is effectively the same as a sequence of statements. Again the loop level to which PICA is applied does not have to have constant bounds as long as it is sufficiently long and the bounds are fixed and known before the loop entry. The profitability of PICA transformation is largely determined by the portion of memory accesses that are offloaded by prefetch, or of the arrays sequentially accessed in the loop level where PICA is applied and below.[8]

An alternative method to apply PICA to the loop nest in Fig. 10(a) is first to switch the order of loops and then to apply PICA to the inner loop. However that solution changes the memory access pattern and therefore may have significant performance impact. In our example the transposed loop has to perform four times as many memory accesses as the original loop (once for each j), and therefore the runtime and energy consumption will greatly increase. Applying PICA directly to an outer loop does not interfere with memory access pattern optimizations that the compiler may have.

8. EXPERIMENTS

We now present our experimental results on the proposed PICA technique. We first describe our evaluation methodology. Next we demonstrate how PICA works through our detailed simulation results, followed by experiments to validate our analytical model, for steady-state as well as transient cases. Further we present our experimental results applying PICA technique to various kernels and nested loops, demonstrating the effectiveness of our technique in reducing the energy of system for memory-bound loops.

8.1. Experimental Setup

To evaluate the performance and energy consumption of PICA technique we use our internally developed and validated XScale simulator. We have modeled this simulator after the 80200 XScale Evaluation Board [Intel Corporation a], which is a full system,

[7] The example in Fig. 10(b) shows the prefetch of "times" array only (line 1), but the other three arrays can be prefetched as well.

[8] We validate the claim on the correlation between profitability and the portion of prefetched memory accesses, through our experiments in Section 8.5.

Table IV. Energy per activity summary

Activity	Energy or Power
Processor power at Run state	450 mW
Processor power at MyIdle state	50 mW
Bus energy per transaction	9.46 nJ
Memory energy per transaction	32.5 nJ

encompassing not only an XScale processor core and caches, but also an AMBA bus and an SDRAM main memory. We have modeled the entire system at the cycle level, up to the behavior of each pipeline stage, and validated it against the 80200 board to be accurate within 7% on average in our cycle count measurements. Our modeling is done in C++ and very extensive, including almost all architectural components such as all functional units, architectural and status registers, all 7 pipeline stages and 12 pipeline latches, dynamic branch predictor, and I- and D-caches with their substructures, plus AMBA bus and memory controller FSM. Our simulator also generates energy estimation for the processor, bus, and memory subsystem. For our study we have extended the simulator to model the prefetch engine and to handle the new PICA instructions.

Another critical element in our evaluation methodology is the XScale compiler, which we have generated from the industry-standard GCC compiler collection version 3.1 [GNU] along with accompanying binary utilities and C library. We have only modified the assembler, which is extended to support the PICA instructions.

For each application we perform PICA transformation manually by adding PICA instructions in inline assembly. Then we compile both the original and the transformed code with high optimization option ("-O3"). The configuration of the system we simulate is as follows. XScale L1 data cache is configured to be 32-way 32K bytes with 32-byte lines unless noted otherwise. The instruction cache has the same architectural parameters. The cache employs the write-back and first-in-first-out policy. The memory bus is 64-bit wide and connected to a single-data-rate SDRAM. We assume the processor-memory clock ratio of 8.

Table IV lists our energy model parameters. The XScale processor can be in either Run state[9] or MyIdle state. A transition between the two states takes 180 cycles, during which the processor is assumed to dissipate an equivalent of the Run state power (450 mW). The MyIdle state is identical to the XScale's IDLE state except that the data cache and the prefetch engine remain active. As in our earlier work [Shrivastava et al. 2005], we assume the data cache power and the prefetch engine power to be 28 mW and about 1 mW, respectively, the latter of which we obtained from our synthesis results of the prefetcher. Thus, together with XScale's memory clock power of 13 mW, the MyIdle power becomes 42 mW, but we conservatively take it for 50 mW. In addition to the processor power we consider memory access energy, which is $(9.46 + 32.5)$ nJ per every transaction, or per every cache line. While we look at dynamic power only, including leakage power will only increase power savings by our technique.

8.2. Simulation Based PICA Exploration

Our deadlock-free PICA technique enables exploration-based optimization of the PICA parameters, T and w. Here we present exploration results to search for the optimal value of T for a fixed value of w.

Figure 11 shows variations in the energy consumption (in mJ) of a type 1 loop with three read references, as T is varied from 50 to 325 iterations. The energy consumption is shown separately for the system, the memory and bus, and the processor core.

[9] If the processor stalls in the Run state, it consumes only 112.5 mW.

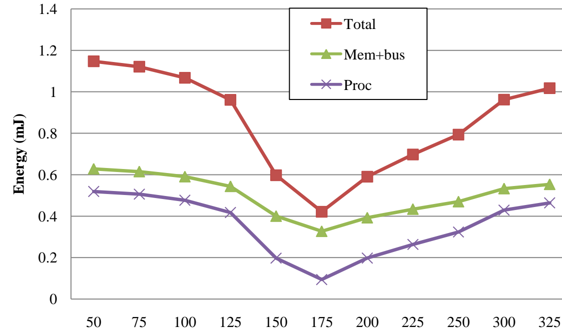


Fig. 11. Energy consumption for different values of T.

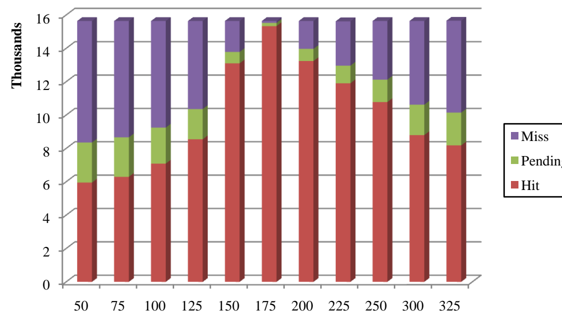


Fig. 12. Cache statistics for different values of T.

Table V. 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

The system energy is minimized at $T = 175$, which is the optimal parameter for this loop from the energy perspective. Compared to the non-PICA case (execution without PICA) the optimized PICA technique reduces the system energy by 47% from 0.79 mJ (not shown) to 0.42 mJ. What is interesting here is that the energy consumption in suboptimal PICA cases can be significantly higher than that of the non-PICA case. This is because in suboptimal PICA cases, the number of external memory accesses can be increased due to some cache lines evicted too early and brought back again on the processor's request. This can be clearly seen in the cache access breakdown in Fig. 12. While the number of total data cache accesses remains the same, the hit and miss ratio varies greatly with different values of T. The increased cache misses translate to increased runtime and energy, which explains the increased energy in the memory and bus subsystem for non-optimized PICA cases.

8.3. Validation of Analytical Model (Steady-State)

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 V lists the main

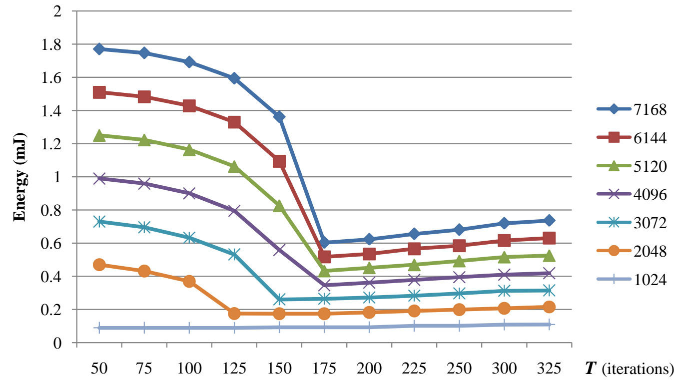


Fig. 13. Energy for different T (type 5).

Table VI. Exploration (with asterisk) vs. analysis results. Energy in mJ.

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%

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 V). 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.

Figure 13 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 VI 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.

Table VII. Validation results varying N to see transient effect

N	#line req.	w	T	T_{act}	w^*	T^*	T^*_{act}
48	54	**54**	(86)	**48**	**50**	(50)	**48**
96	108	**108**	(173)	**96**	**100**	(100)	**96**
192	216	**216**	(346)	**192**	**160**	(200)	**192**
384	432	**256**	(410)	**384**	**240**	(390)	**384**
768	864	**256**	410	**410**	**240**	480	**480**
1536	1728	**256**	410	**410**	**220**	520	**520**

Note: T_{act} is the minimum of T and N. Asterisk indicates exploration results.

Table VIII. Kernel description

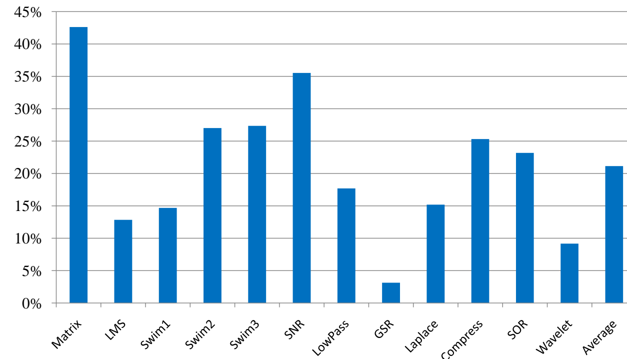
Name	Description	Array Dimension	Type
Matrix	Matrix multiplication	Multi	2
LMS	Least Mean Square	Single	4
Swim1	Weather prediction	Multi	4
Swim2	Weather prediction	Multi	4
Swim3	Weather prediction	Single	1
SNR	Signal-to-Noise Ratio	Single	1
LowPass	Image low pass filter	Multi	3
GSR	Gauss-Seidel Relaxation	Multi	5
Laplace	Edge enhancement	Multi	4
Compress	Image compression	Multi	3
SOR	Succ. Over-Relaxation	Multi	4
Wavelet	Image compression	Single	3

8.4. Transient Effect for Smaller N

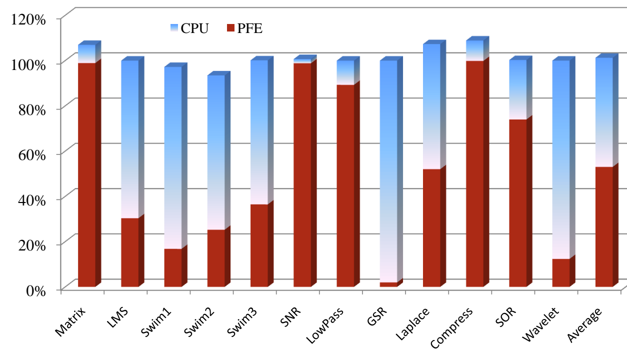
While our analytical model can predict optimal parameters for the steady state, loops in real applications may not reach the steady state. To study the effect of small N, we apply PICA to a matrix multiplication loop and vary the iteration count along with the array size, both of which are denoted by N. We use data cache with 256 lines. Table VII compares the parameters predicted by our model with exploration results. Column 2 lists the maximum number of line requests that can be made per iteration, which is the upper bound of w. Another upper bound of w is the number of cache lines. Since our analytical model does not make use of the iteration count information, the T values may be predicted to be greater than N though any T that exceeds N is effectively reduced to N. T_{act} denotes the reduced value. Through exploration we find the optimal parameters, denoted by w^* and T^*. Again, T^* greater than N should be taken to mean N, which is denoted by T^*_{act}. Comparing w and T_{act} with their asterisk-ed versions indicates that the discrepancy between our analytical model and exploration results is quite limited, even for smaller N. Also, our experimental results confirm once again that analytically found parameters can be good starting points for exploration-based fine tuning.

8.5. 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 [Zivojnovic et al. 1994], SPEC 2000 [SPEC 2000], and multimedia applications. The deadlock-free mechanism allows the PICA technique to be applied to various loops and complex array reference patterns as listed in Table VIII. Eight of them use multidimensional arrays in a non-trivial way. More interestingly, all the kernels we take indeed belong to the first five types in our loop classification as shown in the table, although for this classification we have to exclude the memory accesses that are beyond the scope of prefetching, such as writes, irregular memory accesses, and arrays that are already in the cache before loop execution.



(a) System energy reduction



(b) Memory access profile

Fig. 14. PICA can significantly reduce the energy consumption of various kernels.

For each kernel we optimize the 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 14(a) shows the system energy reduction by our PICA technique over when PICA is not used. The reported system energy includes all the memory accesses (e.g., irregular memory accesses and writes), and we also report the ratio between prefetch engine-issued accesses vs. CPU-issued (e.g., irregular) in Fig. 14(b). 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% (on average 21%) 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.

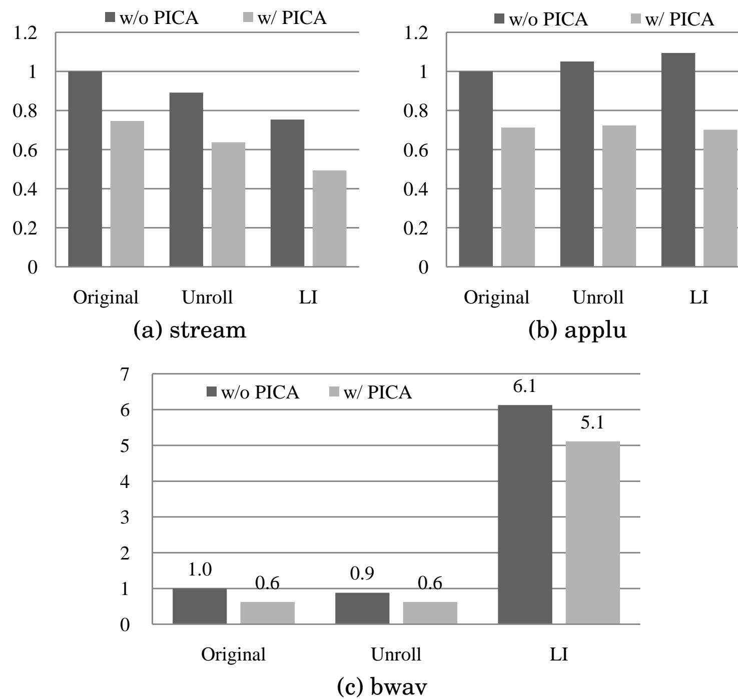


Fig. 15. System energy reduction of nested loops by PICA.

Figure 14(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 Fig. 14(a) and the PFE portion of memory access profile in Fig. 14(b).

8.6. Nested Loop Results

To evaluate the effectiveness of PICA for nested loops we use multiple kernels from the Stream [McCalpin 1995] and SPEC benchmarks. First, Fig. 15(a) shows our simulation results comparing the system energy of the original vs. transformed versions of the stream loop, which are listed in Fig. 10. The y-axis is normalized to the energy consumption of the original loop. Compared to the original (the left-most bar), our PICA transformation (the next one) can achieve about 25% system energy reduction in this case. This reduction is very valuable because if PICA were confined to innermost loops only, the original loop nest wouldn't even have been considered for PICA as it has only four iterations in the innermost loop.

While the innermost loop of Fig. 10 may not be a good candidate for PICA, it certainly is one for loop unrolling. Thus we apply loop unrolling to the original loop (3rd bar in the graph) and compare it with the PICA version (the next bar). Whereas loop unrolling improves energy consumption by 11% in this case, further applying PICA gives 34% energy reduction over the original version.

We also apply loop interchange to the original loop nest, which gives an energy reduction (5th bar) that is even greater than that of applying loop unrolling only, and similar to that of applying nested-loop PICA to the original loop. After loop interchange the loop nest has a new inner loop with a very high trip count, allowing for PICA transformation. Applying PICA further reduces the energy consumption. Combining loop interchange and PICA gives the highest energy reduction of 50% in this case. However, this is not always the case as shown in the other examples.

Fig. 15(b) and (c) show the results of applying the same set of transformations to applu (from SPEC 2000) and bwav (from SPEC 2006), respectively. Applu has a very similar access pattern as the stream loop, but they display very different energy results when loop transformations are applied. One major reason behind their opposite behavior is that whereas the number of data cache misses in stream is almost the same before vs. after loop interchange, in the case of applu, loop interchange increases data cache misses by about 13%—this is all in the absence of PICA transformation. Such different outcomes are very difficult to predict without elaborate analysis or simulation. In all these cases, however, PICA manages to reduce the system energy. This is because PICA, through its own loop restructuring and prefetching, can effectively rearrange the memory accesses in a way that maximizes data reuse in the cache. The last example, bwav, shows an even starker difference. In this case loop interchange destroys data reuse, greatly increasing both execution time and energy consumption of the application, although PICA still manages to reduce system energy consistently compared to without PICA.

As demonstrated by the above experimental results, loop transformations can sometimes worsen the memory access pattern and it is not always obvious to correctly predict what is the best set of loop transformations. Thus although loop transformations can increase the scope of innermost-loop PICA in general, nested-loop PICA further increases the scope and effectiveness of PICA beyond that of innermost-loop PICA.

9. CONCLUSION

In this paper we presented the PICA approach. PICA is a compiler-microarchitecture cooperative technique that can effectively utilize the memory stalls of a processor to achieve low power with little performance overhead. Our enhanced PICA greatly improves the robustness and applicability of processor stall aggregation based energy reduction techniques. First our enhanced PICA guarantees functional correctness for any set of parameters, which facilitates exploration-based parameter optimization. This allows us to fine tune optimal PICA parameters for any memory-bound loop, thus greatly improving applicability of the technique. Second, since exploration based PICA parameter optimization may be time consuming, we developed static analysis for most common types of loops. Third, we extend PICA to multi-nested loops and loops with variable bounds. We discuss the conditions for eligible loops. This makes PICA applicable to a wider class of important applications.

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 intend to research into multi-dimensional prefetch and PICA transformations, which will become necessary in multi-nested loops with complex memory access patterns.

ACKNOWLEDGMENT

This work was supported in part by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by MEST, under grant 2010-0011534, and in part by funding from National

Science Foundation grants CCF-0916652, CCF-1055094 (CAREER), NSF I/UCRC for Embedded Systems (IIP-0856090), Raytheon, Intel, Microsoft Research, SFAz, and Stardust Foundation.

REFERENCES

AZEVEDO, A., ISSENIN, I., CORNEA, R., GUPTA, R., DUTT, N., VEIDENBAUM, A., AND NICOLAU, A. 2002. Profile-based dynamic voltage scheduling using program checkpoints. In *DATE '02: Proceedings of the conference on Design, automation and test in Europe*. IEEE Computer Society, Washington, DC, USA, 168.

BENINI, L., BOGLIOLO, A., AND MICHELI, G. 2000. A survey of design techniques for system-level dynamic power management. *IEEE Transactions on VLSI Systems 8,* 3, 299–316.

BROCKMEYER, E., MIRANDA, M., CORPORAAL, H., AND CATTHOOR, F. 2003. Layer assignment techniques for low energy in multi-layered memory organisations. In *Proc. 6th ACM/IEEE Design and Test in Europe Conf.* Munich, Germany, 1070–1075.

BURD, T. D. AND BRODERSEN, R. W. 2000. Design issues for dynamic voltage scaling. In *ISLPED '00: Proceedings of the 2000 international symposium on Low power electronics and design*. ACM, New York, NY, USA, 9–14.

CHATTERJEE, S., PARKER, E., HANLON, P. J., AND LEBECK, A. R. 2001. Exact analysis of the cache behavior of nested loops. *SIGPLAN Not. 36*, 286–297.

CHOI, K., SOMA, R., AND PEDRAM, M. 2005. 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 Transactions on Computer-Aided Design of Circuits and Systems 24,* 1, 18–28.

GHOSH, S., MARTONOSI, M., AND MALIK, S. 1997. Cache miss equations: an analytical representation of cache misses. In *Proceedings of the 11th international conference on Supercomputing*. ICS '97. ACM, New York, NY, USA, 317–324.

GNU. *GNU Compiler Collection*. GNU. [Online]. Available: http://gcc.gnu.org/.

GOWAN, M. K., BIRO, L. L., AND JACKSON, D. B. 1998. Power considerations in the design of the alpha 21264 microprocessor. In *Design Automation Conference*. 726–731.

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

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

ISSENIN, I., BROCKMEYER, E., MIRANDA, M., AND DUTT, N. 2004. Data reuse analysis technique for software-controlled memory hierarchies. In *Proc. of Design, Automation and Test in Europe*. 202–207.

KANDEMIR, M. AND CHOUDHARY, A. 2002. Compiler-directed scratch pad memory hierarchy design and management. In *ACM/IEEE Design Automation Conference*. 690–695.

LEE, J. AND SHRIVASTAVA, A. 2008. Static analysis of processor stall cycle aggregation. *to appear in* ACM CODES+ISSS. Also available: http://www.public.asu.edu/~ashriva6/papers/pica.html.

LEE, J.-E., KWON, W., KIM, T., CHUNG, E.-Y., CHOI, K.-M., KONG, J.-T., EO, S.-K., AND GWILT, D. 2005. System level architecture evaluation and optimization: an industrial case study with AMBA3 AXI. *Journal of Semiconductor Technology and Science 5,* 4, 229–237.

MCCALPIN, J. D. 1995. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, 19–25.

MOWRY, T. C., LAM, M. S., AND GUPTA, A. 1992. Design and evaluation of a compiler algorithm for prefetching. In *International Conference on Architectural Support for Programming Languages and Operating Systems*. Vol. 27. 62–73.

RABAEY, J. AND PEDRAM, M., Eds. 1996. *Low Power Design Methodologies*. Kluwer Academic Publishers, Norwell, MA.

SHRIVASTAVA, A., EARLIE, E., DUTT, N., AND NICOLAU, A. 2005. Aggregating processor free time for energy reduction. In *CODES+ISSS '05: Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. ACM, 154–159.

SHRIVASTAVA, A., LEE, J., AND JEYAPAUL, R. 2010. Cache vulnerability equations for protecting data in embedded processor caches from soft errors. *ACM SIGPLAN Notices 45,* 4, 143–152.

SPEC 2000. *Standard Performance Evaluation Corporation*. SPEC, http://www.spec.org/.

UNSAL, O. S., KOREN, I., KRISHNA, C. M., AND MORITZ, C. A. 2002. Cool-fetch: Compiler-enabled power-aware fetch throttling. *IEEE Computer Architecture Letters 1*.

VANDERWIEL, S. P. AND LILJA, D. J. 2000. Data prefetch mechanisms. *ACM Computing Surveys 32,* 2, 174–199.

VERDOOLAEGE, S., SEGHIR, R., BEYLS, K., LOECHNER, V., AND BRUYNOOGHE, M. 2007. Counting integer points in parametric polytopes using Barvinok's rational functions. *Algorithmica 48,* 1, 37–66.

ZIVOJNOVIC, V., MARTINEZ, J., SCHLÄGER, C., AND MEYR, H. 1994. DSPstone: A DSP-oriented benchmarking methodology. In *Proc. of ICSPAT '94*.