A Sample-Based Cache Mapping Scheme

Rong Xu
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907
xur@cs.purdue.edu

Zhiyuan Li
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907
li@cs.purdue.edu

Abstract

Applications running on the StrongARM SA-1110 or XScale processor cores can specify cache mapping for each virtual page to achieve better cache utilization. In this work, we describe a method to efficiently perform cache mapping. Under this scheme, we select a number of loops for sampling. These loops are selected automatically based on clock profiling information. We formulate the optimal cache mapping problem as an Integer Linear Programming (ILP) problem. Experiments performed on 14 test programs show speedups in 13 of them (over the default mapping) after applying our sample-based cache mapping scheme. The geometric mean of program speedups for all the 14 test programs is 1.098. Furthermore, compared with a previous heuristic method which uses the full memory trace, the sample-based method performs cache mapping faster by an order of magnitude without sacrificing the quality of mapping.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Optimization

General Terms Performance

Keywords handheld devices, cache mapping, mini cache, cache bypass, profiling, trace sampling

1. Introduction

As the gap between processor and memory speed continues to widen, cache performance becomes increasingly critical to the overall system performance. Researchers have observed that different program regions may have different access patterns and reuse types, and it is possible to fine-tune the cache management mechanism to exploit the reuse behaviors and achieve a better memory performance [4, 2, 3]. The Intel StrongARM SA-1110 [8] and the Intel XScale [9] processor cores have two data caches, namely the main data cache and the smaller-sized mini cache, at the same level in the cache hierarchy. We can map virtual pages to either one of the two caches to get better overall cache reuse. The cache scheme in these processors also supports cache bypass. Cache bypass reduces the amount of data fetched from the main memory by not caching the data item which exhibits low locality. In addition, by keeping non-reusable data items out of the cache, cache bypass also leaves more cache space for the reusable data.

In similar cache schemes proposed previously [17, 10, 5], the cache selection and the bypass decision are made by the hardware. In contrast, on the StrongARM SA-1110 and the XScale processors, the program specifies explicitly for each virtual page whether it should bypass the cache and, if not, which cache it should use. We call the process of specifying the mapping between the virtual pages and the caches *cache mapping*. In our previous work [22], we evaluated the cache system in the StrongARM SA-1110 processor core. We formulated cache mapping as an optimization problem and proved it to be NP-hard. A heuristic based on a full-trace analysis was proposed to derive a cache mapping which is better than the system's default in which all the virtual pages are mapped to the main cache. Unfortunately, the heuristic based on the whole trace is too time consuming for practical use. In the experiments, the heuristic takes 5 minutes to 7 hours for traces ranging from 29.8 MB to 503 MB.

To use cache mapping to improve program memory performance, we must have a way to estimate the dynamic memory access behavior of the program. Typical applications which run on the StrongARM and the XScale processors perform multimedia tasks, encryption, and compression, among others. The dynamic constructs in these programs (pointers, indirectly accessed arrays, and branches) make it difficult to predict the runtime memory behavior through static program analysis only. Therefore, we use profiling to obtain important runtime information. We use a training set of input to find a cache mapping, which is also applied to other runs whose inputs may be different. In practice, one needs to obtain profiling information using the training set in a timely manner in order to try several training data sets and compute the results. Therefore, we must find a cache mapping scheme which is much faster than our previous heuristic method, but without sacrificing the mapping quality.

In this paper, we propose a sample-based, new mapping scheme to achieve the objective stated above. Under this scheme, a number of important loops are automatically selected for sampling. The optimal cache mapping for the simplified memory trace is obtained by solving an Integer Linear Programming (ILP) problem. In our experiments performed on 14 test programs, we find 13 programs to show a performance improvement (over the default mapping) after applying our sample-based cache mapping scheme. The geometric means of program speedup is 1.098. Furthermore, the sample-based method performs cache mapping faster by an order of magnitude than a previous heuristic method using the whole memory trace, without sacrificing the mapping quality.

The rest of the paper is organized as follows. We briefly review the background of the work in Section 2. Our sampling framework is presented in Section 3. Section 4 discusses a method to choose

LCTES'05, June 15–17, 2005, Chicago, Illinois, USA.

$$
\begin{aligned}
T &= T_{miss} * (N_{miss\_main} + N_{miss\_mini}) \\
&\quad + T_{hit} * (N_{hit\_main} + N_{hit\_mini}) \\
&\quad + T_{noncacheable} * N_{noncacheable}
\end{aligned}
$$

where

T_{hit}	:	access time for a cache hit,
T_{miss}	:	average access time for a cache miss,
$T_{noncacheable}$	:	access time for a noncacheable access,
N_{main}	:	total number of accesses to the main cache,
N_{mini}	:	total number of accesses to the mini cache,
$N_{noncacheable}$	:	total number of noncacheable accesses.

Table 1. The formula to compute the total memory access time

the sampled loops. In Section 5, we describe the formulation of the optimal cache mapping problem as an ILP problem. Section 6 shows the experimental results and Section 7 discusses related work. We conclude the paper in Section 8.

2. Cache Mapping

In this section, we first give an overview of the cache system in the Intel StrongARM SA-1110 processor core. We then define the cache mapping problem. We also discuss the concept of the *population list*, which is an extension of the *reuse distance* [14]. We use this concept to determine the cache mapping in our scheme.

2.1 Cache System in StrongARM SA-1110 Based Processors

The Intel StrongARM SA-1110 processor core[8] employs two logically separate data caches, i.e. the main data cache and the mini cache. The 8K-byte main data cache is 32-way set associative with the round-robin replacement. The 512-byte mini cache is 2-way set associative with the LRU replacement. The cache line size is 32 bytes on both caches. For each data cache access, both caches are probed in parallel. However, a particular memory block can exist in only one of the two caches at any time.

Both the main cache and the mini cache are indexed and tagged by virtual addresses. In cache mapping, all memory blocks in the same virtual page will be mapped to the same cache. The mapping is controlled by the bufferable bit (`B`) and the cacheable bit (`C`) in the page table entry in the MMU. If `B=1` and `C=1`, which is the default, the data in this page go to the main data cache. If `B=0` and `C=1`, the data go to the mini cache. If `C=0`, then the page is noncacheable and the memory accesses bypass both caches. This mechanism provides the compiler or the application programs the ability to control page-to-cache mapping by modifying the `B` and `C` bits in the MMU. Note that we need to flush the caches and the TLB entries for consistency after changing the mapping of virtual pages.

2.2 The Problem

Formally stated, the cache mapping problem is as follows. Let the main data cache size be S, the mini data cache size be S_{mini}. We want to assign each virtual page P_i to one of the three mutually exclusive sets, Set_{mini}, Set_{main}, and $Set_{noncacheable}$, such that the total memory access time, denoted as T, computed as shown in Table 1, is minimized.

2.3 Population Lists

We assume that the caches are fully associative and the replacement policy is LRU. We can use the *reuse distance* [14] to predict cache hits and misses. For each memory access, its reuse distance is the number of distinct locations accessed between the current reference and the most recent reference to the same memory address. If the reference is the first access to that memory location, we let the reuse distance be ∞. Unless stated otherwise, all the memory locations mentioned in this paper are identified by the memory block indices, instead of the addresses. This way, spatial locality will appear as temporal locality, and we can focus our attention to temporal locality only. Assuming a fully associative cache with the LRU replacement policy, the cache hits can be computed based on reuse distances. A memory reference with a reuse distance smaller than the cache size (in the number of cache lines) will be a cache hit. In contrast, a reference with a reuse distance greater than or equal to the cache size will be a cache miss. So the total number of hits are equal to the number of references whose reuse distances are less than the cache size.

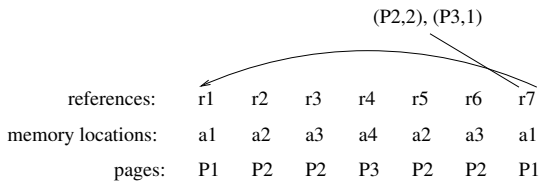

Figure 1. An example of population lists

In the presence of multiple heterogeneous caches at the same level, the exact value of a reuse distance depends on how the pages are mapped to different caches. Given any reference, different cache mapping can produce a different reuse distance, because the reuse distance of a reference is determined by accesses to the same cache. We extend the concept of reuse distances to the concept of *population lists*.

Definition. (*Population lists* [22]): Suppose we are given a set of pages ($\mathcal{P}$) and a sequence of memory references $R = r_1, r_2, \ldots, r_n$. For a reference $r_i \in R$ accessing memory location a, suppose $r_j \in R$ is the reference to the same memory location a such that (1) $j < i$, which means r_j occurs before r_i, and (2) there is no other intermediate access to a. We define the population list of r_i as a list of pairs (k, c_k), where k represents page $P_k \in \mathcal{P}$ referenced between r_i and r_j, and $c_k > 0$ is the number of distinct memory locations in page P_k which are referenced meanwhile.

Figure 1 shows an example of population lists. We can see that between references $r7$ and $r1$, there is no intermediate access to memory location $a1$. Two distinct memory locations in page $P2$ ($a2$ and $a3$), and one memory location in $P3$ ($a4$) are accessed meanwhile. So the population lists for reference $r7$ is $(P2, 2), (P3, 1)$.

To compute the reuse distance of r, under any given page mapping, we just need to add up c_k (the number of distinct memory blocks) for each page k which is mapped to the same cache as the page containing r. In the example shown in Figure 1, if page $P2$ is mapped to the mini cache and page $P1$ and $P3$ are mapped to the main cache, the reuse distance of $r7$ will only count the portion of $P3$ in the population list. Hence we have a reuse distance of 1 for $r7$.

Given a memory trace, population lists can be computed using the same algorithm for computing reuse distances [14], whose complexity is $O(nm)$, where n is the total number of the memory references and m is the number of pages. Note that our method assumes that caches are fully associative with the LRU replacement policy, which is different from the caches in the StrongARM SA-1110 processor. As we showed in our previous work [13], this difference in the cache associativity and the replacement policy

does not have a big impact on the effectiveness of our reuse distance based method.

3. A Sample-Based Framework

In this paper, a *sample* means a single memory access. During the sampling process, we take a consecutive sequence of samples for a period of time. We call such a sequence a *sub-trace*, to distinguish it from the whole memory trace.

Sampling is widely used in performance analysis and program optimization. Different problems may need different methods of sampling. The most common sampling method is *event-based sampling*, also called time-based sampling [12], which takes samples periodically. Unfortunately, time-based sampling creates gaps between memory references which make it unsuitable for making cache-mapping decisions.

A known extension to time-based sampling is to increase the number of samples taken when a sampling event occurs. Instead of taking a single sample, for every n memory accesses, we take $m \gg 1$ consecutive samples. However, making m and n constant has a severe difficulty. The sampling rate is determined by the pair (m, n). Different programs may need totally different values of (m, n) to obtain a representative sub-trace. Obviously, a greater value of m will offer a high accuracy. However, to limit the number of samples to a manageable size, n must be made large for a large m. Unfortunately, a large n may result in a precision loss because it creates large gaps between some samples.

Therefore, in this work, we vary the values of m and n in different parts of the memory trace. We take a large number of samples (by increasing the loop sampling length defined below) for the loops considered important, as explained below. We make the cache mapping decision based on these samples.

We call a loop selected for sampling a *sampled-loop*. For a sampled loop, we define its loop sampling length (*LSL*) as the number of iterations to be sampled. The LSL for each sampled loop must be large enough so that the generated sub-trace from a sampled loop must have the following two properties. (1) In these chosen iterations, there must exist data reuses. (2) These reuses cannot be realized by the default cache mapping where all the pages are mapped to the main cache. A different cache mapping which employs the mini cache and cache bypass may have a better locality for these reuses. We can vary the LSL according to a cost model. Instead of blindly guessing the LSL for each sampled loop, we use profiling information to select the sampled loops and their LSLs as will be discussed in Section 4. The cache mapping, however, is global, which affects all the loops in the program. We use an ILP solver to find the optimal mapping given the information from the samples, i.e. to find a cache mapping that has the minimum average memory access time for the whole program.

In the remains of this section, we describe our sampling framework. How to choose the sampling loop and its LSL will be discussed in Section 4.

3.1 The Sampled Data Objects

References to scalars and small objects usually do not access a large number of memory locations, and their cache hit ratio is usually very high. In contrast, accesses to large arrays tend to spread over a large number of memory locations, and they are the main source of cache misses. Therefore, we sample large arrays only. In real C programs, many array references may take the form of pointer dereferences. We perform a point-to analysis in the compiler (GCC) to find the *point-to set*. If a pointer points to a large array, then we sample memory accesses which are caused by the pointer dereferences. In the sampling run, we discard those accesses that fall out the address space of the target array. The cache mapping in the StrongARM SA-1110, however, is in the unit of a virtual page. To avoid the unnecessary conflict between sampled objects and un-sampled objects, we transform the array declaration such that the arrays are page aligned.

3.2 The Main Steps

Figure 2 shows the overview of the sampling framework. The goal of the process is to find a cache mapping which reduces program's average memory access time.

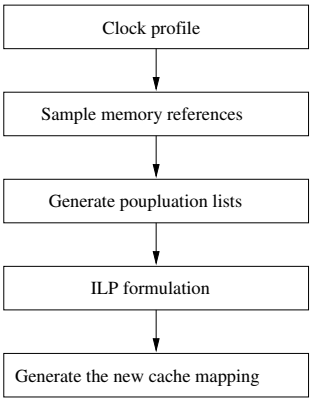

Figure 2. Framework

In the first step, we apply clock profiling to the target program to find the sampled-loops and their LSLs. Detail of this step is describe in the next section.

After that, we instrument the sampled-loops to generate the sub-traces. Let n be the LSL of the sampled loop, the sub-trace consists of the memory accesses to the target objects in n consecutive iterations of this loop.

In the third step, we generate the population list for each sub-trace. In the next step, an ILP problem is formulated based on the population lists whose details are described in Section 5. After solving the ILP problem, we obtain a cache mapping which decides for each virtual page whether it should bypass the caches and if not, which cache it is assigned to. We map the virtual pages back to the data object and insert system calls in the original program to map different parts of the data object to different caches according to the cache mapping. Note that we do not assume that arrays are all statically allocated. The sub-trace includes all memory references. During the trace analysis, memory addresses are used to identify data objects in the program. Page mapping decisions are remembered for the individual pages within each data object using their relative positions. At run time, page mapping is done by system calls using the virtual page numbers available after dynamic memory allocation.

4. Profile-Guided Trace Sampling

In order to reduce the sample size, we first identify those important loops in a given program. We regard a loop to be important if it contains data reuses that are not realized by the default mapping. Recall that the default decision is to map all data to the main cache only. For each loop, we estimate the number of memory blocks it accesses. If the number of memory blocks is greater than the number of cache lines, the data set of this loop does not fit in the cache. If there exist data reuses in this loop, then we may realize at least a subset of these by utilizing the mini cache or making some references bypass the caches. We sample all such important loops. If the virtual pages referenced in the important loops are also referenced in other loops, we include all those loops for sampling.

4.1 Sampled Loops and Their LSL

To choose the loops for sampling and to determine the number of iterations to be sampled (LSL), we apply the concept of footprints and dependence distance vectors [21] to estimate the degree of reuses in each loop. The footprint of a loop is the number of memory blocks which are accessed during the execution of the loop. The dependence distance vector of a loop describes the difference vector between the iterations containing the source and the sink of dependence. Note that the dependence here includes input dependences [21].

Given a loop nest, we compute the dependence distance vectors for all the arrays and the coalesced dependence distance [19] for each dependence distance. The coalesced dependence distance is defined as follows: For a loop nest, suppose b_k is the invariant trip count of loop at level k. Let the coalescing vector of the given loop nest be $\vec{s} = (s_1, s_2, ...s_n)$ such that $s_n = 1$, $s_k = s_{k+1}b_{k+1}$, $1 \leq k \leq n-1$. We define the coalesced dependence distance of dependence vector $\vec{d}$ as the inner product $\vec{d}\vec{s}^T$.

Let $\vec{d} = \{d_1 \dots d_n\}$ be the dependence distance with the longest coalesced distance, and d_i be the first non-zero entry in $\vec{d}$. We compute F, which is the footprint of d_i iterations, as discussed in Section 4.4, and we use the following criteria to choose the sampled-loops and their LSLs.

- If F is greater than the cache size, we make LSL $= R * d_i$, where constant R is chosen as described below.
- If F is less than the cache size, we first determine K, the least number of iterations which have a footprint of size F, and make the LSL $= R * K$.

We extrapolate the reuse pattern in the sample iterations to the entire set of iterations. The number of reuses for the whole trace (r_t) is estimated from the number of iterations of the loop (n), the number of iterations in the sub-trace (s), and the number of reuses in the sub-trace (r_s) by the formula $r_t = \frac{n}{s} * r_s$. We want to choose R such that the estimated number of reuses is close to r_t. To better illustrate the idea, we use Figure 3 to explain how we decide R. Note that although the following calculation is for a two-level nested loop, the idea can be applied to any loop nest. Figure 3 shows the iteration space of a two-level nested loop and a dependence distance of $\vec{d} = (d1, d2)$. The number of reuses for the whole iteration space due to $\vec{d}$ is $r_t = n*m-d1*m-d2*n$. The number of reuses for the sub-trace is $r_s = R*d1*m-d1*m-d2*R*d1$. So the estimated number of reuses for the whole loop is

$$(R*d1*m-d1*m-d2*R*d1)*n/(R*d1) = n*m-m*\frac{n}{R}-d2*n.$$

Since we usually have $n \gg d1$, a large R will make our calculation close to the real number of reuses. On the other hand, a larger R will increase the size of sub-trace. In our experiment, we set $R = 16$ to balance the precision and the cost.

Take matrix-multiply in Figure 4 as an example. The dependence distance with the longest coalesced distance in this loop nest is $\vec{d} = (1, 0, 0)$. The footprint of one iteration of I-loop is $N*N/8+N/8+N/8$, assuming the cache line size is 4 integers. If the main cache size is less than $N*N/8+N/4$, the LSL of this loop nest is 16 for the i-loop.

Due to many reasons, such as random array accesses or indirect array accesses, one may be unable to compute the data dependence distance vector at compile time. In such cases, we assume that every reference accesses a different element of the accessed arrays and hence the longest coalesced distance equals the total sizes of the accessed arrays. Although this estimation is coarse, it errs on the safe side by overestimating the LSL.

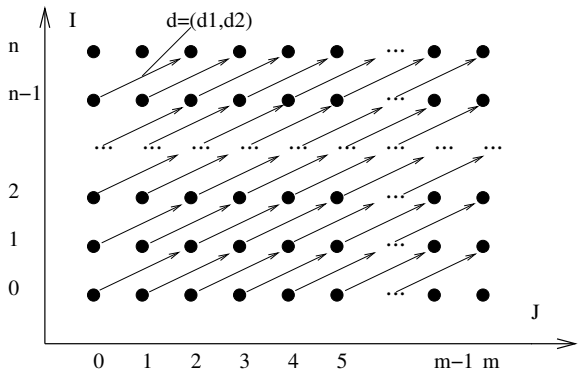

Figure 3. The number of reuse due to the dependence.

```
int a[N][N], b[N][N], c[N][N];
...
for (i=0;i<N;i++)
    for (j=0;j<N;j++) {
        c[i,j] = 0 ;
        for (k=0;k<N;i++)
            c[i][j] += a[i][k] * b[k][j];
    }
```

Figure 4. A loop nest for matrix multiplication

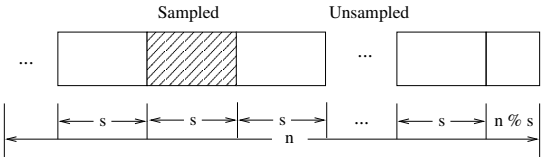

Figure 5. Project the mapping to un-sampled pages.

4.2 Extrapolating the Mapping

The sub-trace may not cover all the virtual pages accessed in the whole loop. As shown in Figure 5, the number of virtual pages accessed in the sampled loops (called the *sampled virtual pages*) and the total number of pages are s and n, respectively. Our ILP formulation in the later stage will only capture the reuses within the sub-trace. The reuse pattern, in most cases, will be repeated in the un-sampled region. As a result, we extrapolate the page mapping to the un-sampled pages for each data object. We divide the pages in each data object into chunks of size s each of which has the same cache mapping as the sampled virtual pages. Let $map(P_i)$ be the mapping decision for page P_i, where i is the page index. Suppose the starting page of the data object is P_t, and the sub-trace includes the memory access from page P_j to page P_{j+s}, where $s > 0$. We have

$$mapping(P_i) = \begin{cases} mapping(P_{(i-j)\%(s+1)+j}) & \text{for } i = j+s+1 \dots t+n \\ mapping(P_{(j-i)\%(s+1)+j}) & \text{for } i = t \dots j-1 \end{cases}$$

To facilitate the computation of LSLs, we first build a reference graph to show the loop nesting relation in the program.

4.3 The Reference Graph

We aim to sample the references inside loops which consist of the majority of the array references. We use a reference graph, $G = (V, E)$, to represent reference relations in the program. Each

node $v_i \in V$ may represent either a loop, a function, or a memory reference to the target arrays. We define quantity $n(v_i)$ for each node in the graph:

- If v_i is a loop node, then $n(v_i)$ is the loop trip count;
- If v_i is a function node, then $n(v_i)$ is the number of times this function is called;
- If v_i is an array reference node, then $n(v_i)$ is the number of instances of this reference (i.e. number of memory accesses).

If v_j is in the body of v_i, we draw an edge $e_{i,j} \in E$ from v_i to v_j. We use a quantity $n(e_{i,j})$ to represent the number of instances of v_j that is executed for one instance of v_i on average. The reference graph may expand cross functions. Since each function corresponds to one node, the graph is context insensitive. Note that a function may be invoked from different call sites and the code represented by a node may be invoked in different loop iterations. The corresponding node in the reference graph may therefore have a varying number of instances. In this case, we take the average number. We obtain the values of $n(v_i)$ and $n(e_{i,j})$ from profiling. Figure 6 shows part of the reference graph for program PEGWIT/encrypt. In this program, we choose to sample array logt and expt. The number in each vertex v_i within a pair of parenthesis is $n(v_i)$. The number marked on each edge is $n(e_{i,j})$.

Note that if there exist no recursive functions in the program, the reference graph is a DAG. For those program with recursive functions, we can eliminate the recursive back edges in the graph by introducing a pseudo loop node. Suppose node v_1, v_2, ... , v_t are in the recursive functions, and $e_{t,1}$ is the back edge. We add a pseudo node v_p to the reference graph with $n(v_p) = n(v_t)n(e_{t,1})$. We then eliminate any edge which exists between node v_1, v_2, ... , v_t. Finally, for each node v_i where $1 \leq i \leq t$, we add an edge $e_{p,i}$ with $n(e_{p,i}) = n(v_i)/n(v_p)$ to the graph. Figure 7 shows an example of this transformation. In the rest of this paper, we assume that the reference graph is a DAG.

4.4 Estimating the Footprint

It is often difficult for the compiler to determine the exact footprint. We estimate the footprint by following the formulas in Table 2. In this table, we compute the footprint for each node through a backward traversal of the reference graph. We use the number of accesses together with the amount of dependence to approximate the size of the accessed array region. Since the number of distinct array elements accessed cannot exceed the number of element of the array, we limit the footprint by sizes of arrays. $accessArray(v_i, A)$ denotes the number of memory accesses to array A in a single instance of v_i; $accessIter(v_i, k)$ denotes the number of memory accesses to all arrays in k iterations of v_i; $footprintArray(v_i, A)$ denotes the footprint in one iteration of v_i attributed to array A; $footprintArrayIter(v_i, A, k)$ denotes the footprint in k iteration of v_i with respect to array A; $footprintIter(v_i, k)$ denotes the footprint in k iteration of v_i; and $arraySize(A)$ is the size of array A.

5. The ILP Formulation

Although the cache mapping problem is NP-hard [22], after we use sampling to reduce the number of population list, we can afford to use an ILP solver to obtain the optimal solution based on the samples. In this section, we show that, given a set of subtraces, each with a different sample rate, how to formulate the ILP problem.

5.1 Generating the Population List

The first step of the formulation is to generate the population lists for each sub-trace. The sampling substantially reduces the

```
gfInvert(...) {
        ...
        for (;;) {
             ...
             gfSmallDiv(...); ...
        }
        ... = logt[] - logt[];
        ... = expt[];
        gfAddMul(...);
        gfAddMul(...);
        for (;;) {
             ...
             gfSmallDiv(...); ...
        }
        ... = logt[] - logt[];
        ... = expt[];
        gfAddMul(...);
        gfAddMul(...);
}
gfAddMul(...) {
        = logt[]; ...
        for( ... )
             if ( ... = logt[] )
                  ... = expt[];
        ...
}

gfAddMul(...) {
        = logt[]; ...
        for( ... )
             if ( ... = logt[] )
                  ... = expt[];
        ...
}
```

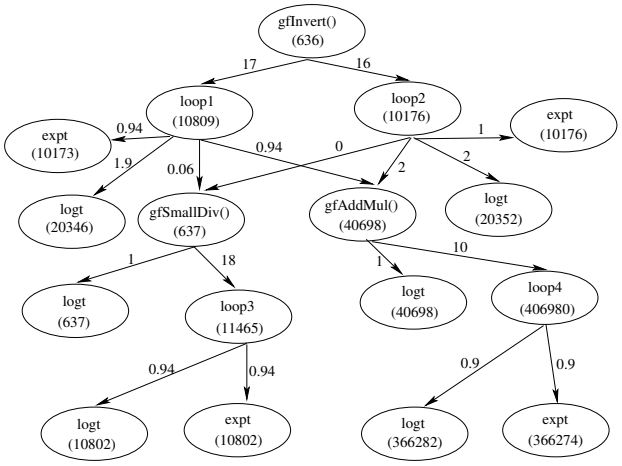

Figure 6. An example of reference graph

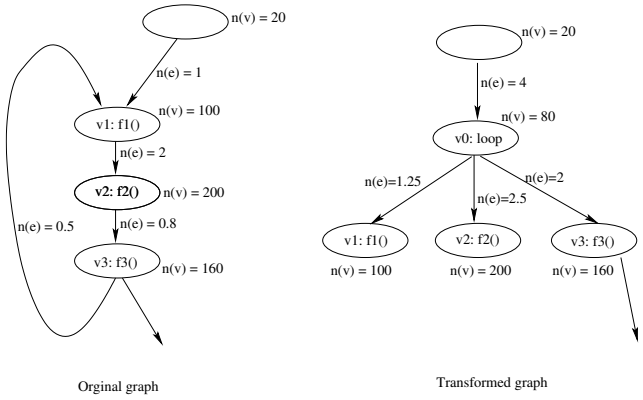

Figure 7. Transform recursive function nodes to a loop node

$\text{accessArray}(v_i, A) =$	$\begin{cases} 1 & v_i \text{ is a reference to A} \wedge v_i \text{ is a leaf node} \\ 0 & v_i \text{ is a reference to other arrays} \wedge v_i \text{ is a leaf node} \\ \sum_{\forall e_{i,j}} \text{accessArray}(v_j, A) & v_i \text{ is not a leaf node} \end{cases}$
$\text{accessIter}(v_i, k) =$	$\sum_{\forall A} k * \text{accessArray}(v_j, A)$
$\text{footprintArray}(v_i, A) =$	$\text{MIN}(\text{arraySize}(A), \text{accessArray}(v_i, A))$
$\text{footprintArrayIter}(v_i, A, k) =$	$\text{MIN}(\text{arraySize}(A), (k-n) * \text{accessArray}(v_i, A))$ n is the amount of dependences with footprints $\leq$ k
$\text{footprintIter}(v_i, k) =$	$\sum_{\forall A} \text{footprintArrayIter}(v_j, A, k)$

Table 2. Equations to calculate foot-print in the reference graph

number of the memory accesses. The sampled trace, however, may produce many identical population lists. As we will show later in this section, each population list introduces a number of constraints and variables to the ILP system. We group the identical population lists to remove the redundant constraints. Our experiments show that this simple strategy is very effective. We reduce the number of population lists by 60% to 98% for our test programs. The time to perform the grouping is in the order of $O(n^2)$, where n is the size of the entire population lists. In practice, since there are many identical population lists, the execution time of grouping is much less than the theoretical upper bound.

5.2 Formulation

The input of the formulation is a set of sub-traces. After generating the population lists, each sub-trace s_t contains the following information:

- The trip count of the sampled loop: $\text{tripCount}(s_t)$.
- The number of iterations we sampled: $\text{nIter}(s_t)$.
- The population lists. For each population list a, we use $\text{nDup}(a)$ to denote the number of population lists which are identical to a.

For a sub-trace s_t, the total memory access time $T(s_t)$ in Table 1 can be rewritten as

$$\begin{aligned} T(s_t) \quad = \quad & T_{miss} * (N_{miss\_main} + N_{miss\_mini} + N_{hit\_main} \\ & + N_{hit\_mini} + N_{noncacheable}) \\ & - T_{miss} * (N_{hit\_main} + N_{hit\_mini}) \\ & - T_{miss} * N_{noncacheable} \\ & + T_{hit} * (N_{hit\_main} + N_{hit\_mini}) \\ & + T_{noncacheable} * N_{noncacheable}. \end{aligned}$$

Let the total number of memory accesses be $L(s_t)$. We have

$$\begin{aligned} T(s_t) = \\ & L(s_t) * T_{miss} - (T_{miss} - T_{hit}) * (N_{hit\_main} + N_{hit\_mini}) \\ & - (T_{miss} - T_{noncacheable}) * N_{noncacheable}. \end{aligned}$$

$T(s_t)$ only takes account of the memory accesses in sub-trace s_t. We estimate the memory access time for the loop from which the sub-trace s_t is taken to be $T(s_t) * \text{tripCount}(s_t)/\text{nIter}(s_t)$. Thus the total memory access time for the program is estimated as

$$T = \sum T(s_t) * \text{tripCount}(s_t)/\text{nIter}(s_t).$$

We have two types of constraints. Each page has a *page-mapping constraint* which guarantees that the page will have a unique mapping decision. Each population list has a number of *population-list constraints* which ensure that only those memory accesses with a reuse distance (under the new cache mapping) less than the cache capacity can be cache hits. The objective function of the ILP program is to minimize T defined above. In the following discussion, we show the details of the ILP formulation.

5.2.1 Page-mapping constraints

For each page P_i, we define three 0-1 variables $p1_i$, $p2_i$ and $p3_i$, and we impose the following constraint:

$$p1_i + p2_i + p3_i = 1 \tag{1}$$

This constraint guarantees that page P_i is in exactly one of the three sets. If $p1_i = 1$, page P_i is assigned to the main data cache. If $p2_i = 1$, page P_i is assigned to the mini cache. If $p3_i = 1$, page P_i is marked as noncacheable.

5.2.2 Population-list constraints

For each population list $a_{i,j} = (k_1, c_{k_1})$ ... (k_τ, c_{k_τ}), we define two variables $a1_{i,j}$ and $a2_{i,j}$, to represent the reuse distances in the main cache and the mini cache, respectively.

$$a1_{i,j} = \sum_{k=k_1 \ldots k_\tau} c_k * p1_k \tag{2}$$

$$a2_{i,j} = \sum_{k=k_1 \ldots k_\tau} c_k * p2_k \tag{3}$$

We introduce two 0-1 variables $h1_{i,j}$ $h2_{i,j}$ and the following constraints.

$$h1_{i,j} * a1_{i,j} \leq S - 1 \tag{4}$$

$$h2_{i,j} * a2_{i,j} \leq S_{mini} - 1 \tag{5}$$

$$h1_{i,j} \leq p1_i \tag{6}$$

$$h2_{i,j} \leq p2_i \tag{7}$$

Constraint (4 – 7) imply the following property. For the optimal solution of the ILP problem, we have $h1_{i,j} = 1$ iff the reference represented by $a_{i,j}$ is a hit in the main cache for the optimal cache mapping where the total memory access is minimized. Furthermore, we have $h2_{i,j} = 1$ iff the reference represented by $a_{i,j}$ is a hit in the mini cache for the optimal cache mapping. If both $h1_{i,j}$ and $h2_{i,j}$ are 0 in the optimal solution of the ILP problem, the reference represented by $a_{i,j}$ is not a cache hit in the optimal cache mapping. It might be a cache miss or a cache bypass access, depending on whether page P_i is marked as noncacheable. Note that we cannot have both $h1_{i,j}$ and $h2_{i,j}$ as 1 since the page mapping constraint guarantees that at most one of $p1_i$ and $p2_i$ can be 1.

5.2.3 Linearizing population-list constraints

Constraints (4) and (5) are not linear. We apply the following transformations to linearize them. First, note that the reuse distance cannot be infinitely large except for the first reference to each memory block. We ignore the relatively small number of references which have an infinite reuse distance. For the remaining references, suppose the largest reuse distance value is V and the main cache size is S, let $K_{main} \equiv \lceil V/S \rceil$. We transform Constraint (4) to

$$a1_{i,j} \leq (S-1) * (K_{main} - y1_{i,j}) \tag{8}$$

$$y1_{i,j} \geq (K_{main} - 1) * h1_{i,j} \tag{9}$$

where $y1_{i,j}$ is an integer variable.

We claim that Constraints (8) and (9) are equivalent to Constraint (4) for the following reason:

- Constraint (4) is equivalent to the following.
  - If $a1_{i,j} \leq S$, $h1_{i,j} \in \{0, 1\}$.
  - If $a1_{i,j} > S$, $h1_{i,j} = 0$.
- Constraint (8) and Constraint (9) have the following implication.
  - If $a1_{i,j} \leq S$, to make Constraint (8) valid, we have $y1_{i,j} \in \{0, 1, ..., K_{main} - 1\}$. If $y1_{i,j} \in \{0, 1, ..., K_{main} - 2\}$, we have $h1_{i,j} = 0$; if $y1_{i,j} = K_{main} - 1$, we have $h1_{i,j} = 1$ (Constraint (9)). So the solution space for $h1_{i,j}$ is $\{0, 1\}$.
  - If $a1_{i,j} > S$, to make Constraint (8) valid, we have $y1_{i,j} \in \{0, 1, ..., K_{main} - t\}$ where $t \geq 2$. With Constraint (9), $h1_{i,j}$ is 0.

Note that $y1_{i,j}$ will not appear in the objective function. If there is a solution satisfying Constraint (4), we can find a solution satisfying Constraint (8) and (9) with the same value of the objective function, and vise versa. Therefore, Constraint (8) and Constraint (9) are equivalent to Constraint (4).

Similarly, we transform Constraint (5) to

$$a2_{i,j} \leq (S_{mini} - 1) * (K_{mini} - y2_{i,j}) \quad (10)$$

$$y2_{i,j} \geq (K_{mini} - 1) * h2_{i,j} \quad (11)$$

where $K_{mini} \equiv \lceil V/S_{mini} \rceil$ and $y2_{i,j}$ is another integer variable. Note that S_{mini} is the size of the mini cache.

5.2.4 The objective function

We first show how to compute the $T(s_t)$ within a sub-trace s_t. Since we have $\mathrm{nDup}(a_{i,j})$ population lists in the form of $a_{i,j}$, the number of hits in both the main cache and the mini cache equals $\sum_i \sum_j \mathrm{nDup}(a_{i,j}) * (h1_{i,j} + h2_{i,j})$. The number of noncacheable references equals $\sum_i \sum_j \mathrm{nDup}(a_{i,j}) * p3_i$. Hence, we have

$$\begin{aligned} T(s_t) = \; & L(s_t) * T_{miss} \\ & - (T_{miss} - T_{hit}) * (\sum_i \sum_j \mathrm{nDup}(a_{i,j}) * (h1_{i,j} + h2_{i,j})) \\ & - (T_{miss} - T_{noncacheable}) * (\sum_i \sum_j \mathrm{nDup}(a_{i,j}) * p3_i) \end{aligned}$$

With $T(s_t)$, we can derive the objective function as discussed earlier in this section. The formulated ILP problem is to minimize $\sum T(s_t) * \mathrm{tripCount}(s_t)/\mathrm{nIter}(s_t)$ subject to constraints (1) (6) (7) (8) (9) (10) and (11).

Suppose there are m virtual pages and n population lists. The ILP problem has $3m + 4n$ variables and $m + 6n$ constraints. The objective function has $3n$ terms.

5.3 Classifying the Population Lists

There exist references which are always cache hits, no matter how the other pages are mapped, as long as their accessed pages are not marked as noncacheable. We call such references the *short-references*. For a reference r with a population list of (k_1, c_{k_1}) ... (k_τ, c_{k_τ}), if $\sum_{k=k_1 \ldots k_\tau} c_k < S_{mini}$, where S_{mini} is the size of the mini cache, r is definitely a short-reference. In the ILP formulation for a short-reference $a_{i,j}$ that accesses to page P_i, it is obvious that if $p1_i + p2_i = 1$, we have $h1_{i,j} + h2_{i,j} = 1$. We can eliminate variables $h1_{i,j}$ $h2_{i,j}$, as well as the associated constraints. As a result, for a short-reference, we do not need to record its population list. Instead, we record the number of short-references in each page. Suppose page P_i has $N1(i)$ short-references. The objective function for this portion of references for all pages can be written as:

$$\begin{aligned} & (T_{miss} - T_{hit}) * \sum_i (N1(i) * (p1_i + p2_i)) + (T_{miss} \\ & - T_{noncacheable}) * \sum_i (N1(i) * p3_i) \end{aligned}$$

Our experiment shows that 70% to 92% of references in the test programs are short-references. This method significantly reduces the size of population lists as well as the size of the ILP problem.

Benchmark	Program	Input parameters
Mediabench	adpcm/rawcaudio	<clinton.pcm >out.adpcm
	adpcm/rawdaudio	<clinton.adpcm >out.pcm
	pegwit/encrypt	-e my.pub pgptest.plain pegwit.enc <encryption_junk
	pegwit/decrypt	-d pegwit.enc pegwit.dec < my.sec
—	matrix multiply	(problem size of 397)
—	compress	input.log
—	decompress	-d input.log
SPEC2000	crafty	crafty.in (test input)
	gzip	input.compressed 2
	mcf	inp.in (test input)
OLDEN	treeadd	12
	power	—
	mst	128
	bisort	15000

Table 3. Test programs and input parameters

6. Experimental Results

Our experiments are performed on a Compaq iPAQ 3650 PDA which has a 206MHZ Intel StrongARM SA-1110 processor and 32M RAM. The cache mapping is implemented through system calls to the Linux kernel. We use ILOG/CPLEX [7], a general purpose ILP solver, to solve the formulated ILP problem. Throughout the experiment, we use `GCC` with the `-O3` switch to compile the program, and we do not apply any cache conscious data placement optimization to the test program. Table 3 lists the test programs and their input parameters.

Table 4 compares the time to find the result cache mapping between the heuristic method and the sampling method. The *total time* column in this table includes the time to calculate the population lists *time1* and the time to solve the cache mapping problem *time2*. We see that the sampling scheme shortens the time to find the cache mapping significantly. For most the programs, the sampling method reduces the time by an order of magnitude. For matrix-multiply, we reduce the time from almost 8 hours to 19 minutes. The *variables #* and *constraints #* columns list the number of variables and constraints in the ILP system for each test program, respectively.

We compare the program performance obtained by the whole-trace heuristic method and the sampling method. Figure 8 shows the normalized execution time. The right bars show the performance improvements by the sampling method which ranges from 1% to 20% with the exception of program `bisort` whose performance degrades by about 2%. The geometric mean of program speedups for all the 14 test programs is 1.098. The performance enhancement is quite close to that obtained by the full-length memory-trace analysis which is shown by the left bar.

6.1 Results of Using Non-training Inputs

The performance data of the new cache mapping shown above were obtained by applying the test programs on the same input files as those used for profiling and trace sampling. We also apply the ob-

program	old heuristic			sample-based				
	time1	time2	total time	time1	time2	total time	variables #	constraints #
rawcaudio	3m26s	21m6s	24m32s	0.9s	31.1s	32s	4078	5998
rawdaudio	4m54s	32m50s	37m44s	1.2s	36.4s	38s	4114	6052
encrypt	1m42s	9m35s	11m17s	2.1s	1m32.8s	1m35s	5564	8234
decrypt	1m3s	5m10s	6m13s	2.3s	1m49.9s	1m52s	5620	8318
mm	36m58s	7h5m19s	7h42m	14.5s	18m22.7s	18m37s	7856	11196
compress	55s	4m8s	5m3s	2.3s	1m29.7s	1m32s	5452	7996
decompress	1m4s	5m8s	6m12s	3.7s	3m13.5s	3m17s	5944	8734
crafty	18m46s	4h9m11s	4h28m	11.4s	27m0.5s	27m12s	9515	14157
mcf	16m55s	2h51m27s	3h8m	9.2s	18m22.0s	18m31s	8796	12998
gzip	14m1s	1h58m3s	2h12m	13.4s	22m8.8s	22m22s	9157	13599
treeadd	17m32s	2h34m27s	2h52m	5.5s	4m52.3s	4m58s	6028	9000
mst	5m14s	34m58s	40m12s	2.1s	2m0.8s	2m03s	5745	8523
power	10m12s	1h32m48s	1h42m	8.7s	10m20.1s	10m29s	6913	10205
bisort	6m55s	1h5h1s	1h12m	4.3s	6m13.5s	6m18s	6447	19499

Table 4. Comparison of time spent for finding the cache mapping

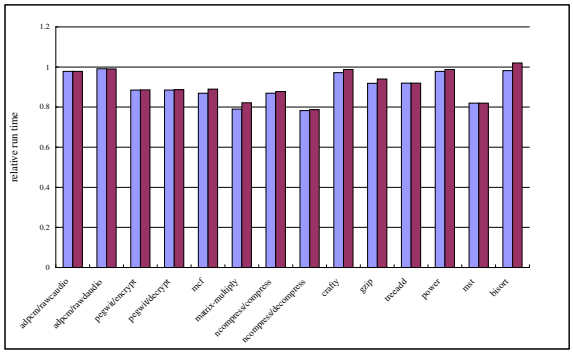

Figure 8. Execution time comparison between cache mapping by the whole-trace method (the left bar) and by the sampling method (the right bar).

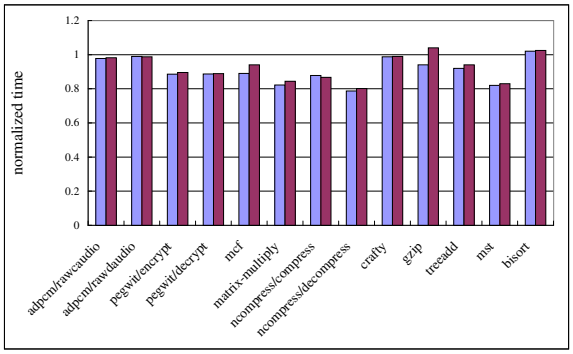

Figure 9. Normalized execution time for different input files using the cache mapping obtained from the sampling framework. The left bar shows the performance using the same inputs while the right bar shows the performance using different inputs.

Program	Input parameters
adpcm/rawcaudio	<small.pcm >out.adpcm
adpcm/rawdaudio	<small.adpcm >out.pcm
pegwit/encrypt	-e my.pub newfile.plain pegwit.enc <encryption_junk
pegwit/decrypt	-d newfile.enc pegwit.dec < my.sec
matrix multiply	(problem size of 697)
ncompress/compress	input.combined
ncompress/decompress	-d input.combined
crafty	crafty.in (train input)
gzip	input.combined 10
mcf	inp.in (train input)
treeadd	20
mst	512
bisort	75000

Table 5. Test programs with different input parameters.

tained mapping to the programs with different input parameters. Figure 9 shows those results and Table 3 lists the used input parameters. Program `power` is omitted from the table because it does not have input parameters. We see that for all the test programs (except `gzip`), the cache mapping obtained from the training input still works when the inputs are different.

7. Related Work

There exist extensive research work on cache bypass and the horizontally partitioned caches [20, 11, 5, 16, 17, 15]. Most of the work, however, uses hardware to control the cache mapping decision. The cache mapping in our study is a software method and thus is different from all the work cited above.

In our previous work [22], we proposed a heuristic which is applied to the whole memory trace to improve cache mapping. The heuristic method is sufficient for the evaluation purpose, but it is not very useful in practice. In this work, using a sampling-based method to obtain the smaller-sized but representative samples, we transform the cache mapping problem to an ILP problem and solve it optimally. This method is faster, by an order of magnitude, than the whole-memory trace method.

Sampling is widely used in program performance analysis [12], feedback-control optimization [1], and dynamic optimizations [18, 6]. Arnold and Ryder [1] propose a framework for fast sampling. Their framework allows functions or loops to switch between the sampling state and the non-sampling state. Hirzel and Chilimibi [6] extend Arnold and Ryder's framework by making it possible to take longer consecutive samples. Since the sampling methods described in these two frameworks are primarily used in online optimization, the focus of their work is to reduce the cost of instrumented code. The user still needs to specify what to sample, where to sample, as

well as the sample rate. In contrast, we focus more on the quality of the sampling. In addition, our framework automatically selects sampled objects, sampled loops, and the sample rates.

Rubin et.al. [18] use memory sampling in data-layout optimization. Their optimization mainly tries to rearrange the layout of data objects in order to reduce the cache conflict misses. The cache mapping optimization does not change the layout of objects and it is aimed at reducing capacity misses by smart page mapping.

8. Summary

In this work, we present an efficient scheme for cache mapping. We propose a framework under which we select a number of loops for sampling. These loops are selected automatically based on clock profiling information. We formulate the optimal cache mapping problem as an Integer Linear Programming (ILP) problem using the sampled trace. Our experimental results show that using this sampling framework is much faster to find the better cache mapping than the whole-trace analysis method. With this fast sampling framework, we also re-evaluate the cache mapping scheme by using different set of inputs for our test programs.

Acknowledgments

This work is sponsored by National Science Foundation through grants CCF-0444285, CCR-0208760, ACI/ITR-0082834, and CCR-9975309.

References

[1] Matthew Arnold and Barbara G. Ryder. A framework for reducing the cost of instrumented code. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 168–179, 2001.

[2] Kennith K. Chan, Cyrus C. Hay, John R. Keller, Gordon P. Kurpanek, Francis X. Schumacher, and Jason Zheng. Design of HP PA 7200 CPU. In *Hewlett-Packard Journal*, February 1996.

[3] C-H. Chi and H. Deitz. Improving cache performance by selective cache bypass. In *the 22nd Hawaii International Conference on System Science*, pages 277–285, January 1989.

[4] D. Chiou, P. Jain, S. Devadas, and L. Rudolph. Dynamic cache partitioning via columnization. In *Proceedings of Design Automation Conference, Los Angeles*, June 2000.

[5] Antonio Gonzalez, Carlos Aliagas, and Mateo Valero. A data cache with multiple caching strategies tuned to different types of locality. In *Proceedings of International Conference on Supercomputing*, pages 338–347, July 1995.

[6] M. Hirzel and T. Chilimbi. Bursty tracing: A framework for low-overhead temporal profiling, 2001.

[7] ILOG Inc. *ILOG CPLEX 7.1 Reference Manual*. 2001.

[8] Intel Corporation. Intel StrongARM SA-1110 microprocessor developer's manual. http://www.intel.com/design/strong/manuals/278 240.htm, October 2001.

[9] Intel Corporation. Intel PXA250 and PXA210 application processor developer's manual. http://www.intel.com/design/pca/applications processors/manuals/278693.htm, February 2002.

[10] Teresa L. Johnson, Daniel A. Connors, Matthew C. Merten, and Wen W. Hwu. Run-time cache bypassing. *IEEE Transactions on Computers*, 48(12):1338–1354, 1999.

[11] Teresa L. Johnson and Wenmei W. Hwu. Run-time adaptive cache hierarchy management via reference analysis. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 315–326, 1997.

[12] Richard E. Kessler, Mark D. Hill, and David A. Wood. A comparison of trace-sampling techniques for multi-megabyte caches. *IEEE Transactions on Computers*, 43(6):664–675, 1994.

[13] Zhiyuan Li and Rong Xu. Page mapping for heterogeneously partitioned caches: Complexity and heuristics. *Journal of Embedded Computing*, accepted.

[14] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM System Journal*, 9:78–117, 1970.

[15] V. Milutinovic, M. Tomasevic, B. Markovic, and M. Tremblay. A new cache architecture concept: the split temporal/spatial cache. In *Proceedings of 8th Mediterranean Electrotechnical Conference*, pages 1108–1111, May 1996.

[16] Jude A. Rivers and Edward S. Davidson. Reducing conflicts in direct-mapped caches with a temporality-based design. In *Proceedings of the 1996 International Conference on Parallel Processing*, volume 1, pages 154–163, 1996.

[17] Jude A. Rivers, Edward S. Tam, Gary S. Tyson, Edward S. Davidson, and Matt Farrens. Utilizing reuse information in data cache management. In *Conference proceedings of the 1998 international conference on Supercomputing*, pages 449–456. ACM Press, 1998.

[18] Shai Rubin, Rastislav Bodik, and Trishul Chilimbi. An efficient profile-analysis framework for data-layout optimizations. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 140–153. ACM Press, 2002.

[19] Yonghong Song, Rong Xu, Cheng Wang, and Zhiyuan Li. Improving data locality by array contraction. *IEEE Trans. Computers*, 53(9):1073–1084, 2004.

[20] Gary Tyson, Matthew Farrens, John Matthews, and Andrew R. Pleszkun. A modified approach to data cache management. In *Proceedings of the 28th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 93–103, 1995.

[21] Michael Joseph Wolfe, Carter Shanklin, and Leda Ortega. *High Performance Compilers for Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[22] Rong Xu and Zhiyuan Li. Using cache mapping to improve memory performance of handheld devices. In *Proceedings of the 4th IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS-2004)*, pages 115–122, Austin, Texas, 2004.