Using Cache Mapping to Improve Memory Performance of Handheld Devices

Rong Xu Zhiyuan Li
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907
{xur,li}@cs.purdue.edu

Abstract

Processors such as the Intel StrongARM SA-1110 and the Intel XScale provide flexible control over the cache management to achieve better cache utilization. Programs can specify the cache mapping policy for each virtual page, i.e. mapping it to the main cache, the mini-cache, or neither. For the latter case, the page is marked as noncacheable. In this paper, we use memory profiling to guide such page-based cache mapping. We model the cache mapping problem and prove that finding the optimal cache mapping is NP-hard. We then present a heuristic to select the mapping. Execution time measurement shows that our heuristic can improve the performance from 1% to 21% for a set of test programs. As a byproduct of performance enhancement, we also save the energy by 4% to 28%.

Key words – handheld devices, cache mapping, multiple caches, mini-cache, cache bypass

1 Introduction

The issue of reducing the average memory access time continues to receive wide-spread attention. One of the hardware approaches proposed in recent studies [20, 15, 24, 5] uses horizontally partitioned data caches. This approach maintains multiple data caches at the same level in the cache hierarchy. Different caches may have different structures. There are several advantages to this approach:

- Different memory addresses may exhibit different locality behaviors and some may have no locality at all. By carefully mapping different data to different subcaches, we may get a higher overall cache hit ratio.
- Smaller subcaches allow a faster CPU clock because of the shorter cache hit time.
- On a partitioned cache, it is possible to probe just one of the subcaches during a data access. This can result in a substantial energy saving [15, 24, 1, 14], which is especially important to handheld devices and embedded systems.

Cache bypass is another technique to reduce the average memory access time. It keeps non-reusable data items out of the cache in order to use the cache space to retain reusable data. For the data items exhibiting low locality, cache bypass also reduces the amount of data fetched from the main memory, because only the target data item, instead of the whole cache line, needs to be transferred. A number of hardware solutions [13, 23] have been proposed to monitor the memory access patterns and to make bypass decisions.

Most of the hardware schemes mentioned above need relatively expensive dedicated hardware, such as extra tags [1], a *nontemporal data detection unit* [20], or a *Memory Address Table (MAT)* [13], to detect memory access patterns. The cost of the hardware prevents these schemes from being used in the processors for current handheld devices and embedded systems.

Processors such as the Intel StrongARM SA-1110 [10] and the Intel XScale [11] use a less expensive technique by allowing application programs or compilers to specify the cache mapping policy for each virtual page. The processor contains a relatively small-sized mini-cache in parallel with the main data cache. Each page can be mapped to either the main cache or the mini-cache, or marked as noncacheable (for cache bypass). Intel Developer's Manual [10] states that the mini-cache is designed to prevent thrashing on the main data cache. Its typical use is to store large data structures such that accesses to these data structures do not interfere with the data in the main data cache.

The support for multiple cache policies provides the potential benefits of both the horizontally partitioned data cache and the bypass cache. However, to take advantage of this feature, one must carefully specify the cache policy

for the individual virtual page. We call the process of specifying the mapping between the virtual pages and the caches *cache mapping*. Although Intel gives guidelines for cache mapping as mentioned above, applying them to real programs faces several challenges: It is often difficult to predict the cache reuse pattern in non-numerical programs. Even the programmer may not predict the cache behavior accurately. Moreover, the decision on cache mapping cannot be made for a single data object in isolation without considering other objects stored in the same page.

In this paper, we use memory profiling to study page-level cache mapping. We model the cache mapping problem and prove that its optimal solution is NP-hard. We present a cache mapping heuristic and implement it for Compaq iPAQ 3650 handheld devices which use the Intel StrongARM SA-1110 processor.

The rest of this paper is organized as follows: in Section 2, we briefly review the cache system in Intel StrongARM SA-1110 and discuss why a cache system with multiple mapping policies can perform better than traditional caches. In Section 3 we formalize the cache mapping problem and prove it to be NP-hard. Section 4 gives a heuristic algorithm. The experimental results are presented in Section 5. We review related work in Section 6 and conclude in Section 7.

2 Background

The Intel StrongARM SA-1110 processor [10] 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 Round-Robin replacement. The 512-byte mini-cache is 2-way set associative with 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. 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 page is mapped to the main data cache. If `B=0` and `C=1`, it is mapped to the mini-cache. If `C=0`, then the page is noncacheable and its 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 page mapping.

By carefully mapping different references to different caches, we can achieve better cache utilization than traditional caches through better cache replacement. Assuming all the caches are fully associative with LRU replacement, consider the following trace where each variable represents a memory block: `x0 x1 x1 x2 x2 x0`. If we have a single cache of size smaller than or equal to 2 cache lines, then a capacity miss will occur at block `x0`. In contrast, if we have two independently indexed caches, each of which has one cache line, then we can allocate `x0` to one cache and `x1` and `x2` to the other cache. No capacity miss occurs.

The idea of reducing capacity misses by selecting memory references to bypass the cache can be illustrated via the following example: `x0 x1 x2 x0`. In this memory trace, there are no reuses for blocks `x1` and `x2`. If we let the references to these blocks bypass the cache, then `x0` can be reused for a cache size as small as one cache line.

3 The Optimal Cache Mapping

We define the CACHE-MAPPING problem as the following: given a memory trace, determine the best page-to-cache mapping such that the average memory access time is minimized. Since we do not reduce the number of references, the objective of minimizing the average memory access time is the same as minimizing the total memory access time, which can be expressed by the following formula:

$$\begin{aligned} T = \quad & T_{hit\_in\_main} * N_{main} * h_{main} + \\ & T_{miss\_in\_main} * N_{main} * (1 - h_{main}) + \\ & T_{hit\_in\_mini} * N_{mini} * h_{mini} + \\ & T_{miss\_in\_mini} * N_{mini} * (1 - h_{mini}) + \\ & \sum_{i=1}^{N_{noncacheable}} (x + S_i/B) \end{aligned} \tag{1}$$

where $T_{hit\_in\_main}$ and $T_{hit\_in\_mini}$ denote access time for a cache hit at the main cache and the mini-cache respectively. $T_{miss\_in\_main}$ and $T_{miss\_in\_mini}$ denote the average access time for a cache miss at the main cache and the mini-cache respectively. The term x is the delay for accessing the first byte of any data in the main memory, B the memory bus bandwidth, and S_i the data size for the i^{th} noncacheable memory access. N_{main} and N_{mini} are the total number of accesses to the main cache and the mini-cache. $N_{noncacheable}$ is the number of noncacheable accesses. h_{main} and h_{mini} represent the hit ratios for the main cache and the mini-cache.

Formally, the page-to-cache mapping is done by assigning each virtual memory page to one of the three mutually exclusive sets, Set_{main}, Set_{mini} and $Set_{noncacheable}$. Set_{main} contains the pages mapped to the main cache; Set_{mini} contains the pages mapped to the mini-cache and $Set_{noncacheable}$ contains the noncacheable pages.

We make the following assumptions to simplify the CACHE-MAPPING problem:

- $T_{hit\_in\_main} = T_{hit\_in\_mini}$ and $T_{miss\_in\_main} = T_{miss\_in\_mini}$. Hence, we simply use the terms T_{hit}

and T_{miss}, respectively. The StrongARM SA-1110 processor probes both caches in parallel, so it is necessary to have $T_{hit\_in\_main} = T_{hit\_in\_mini}$. Since the main memory operation and the bus transmission constitute the main part of the cache miss penalty, $T_{miss\_in\_main}$ and $T_{miss\_in\_mini}$ are approximatively equal.

- All the data items targeted by noncacheable accesses are of the same size. This assumption is reasonable for the SA-1110 processor even though the memory system supports accesses in *burst* mode. This is because, in compiler-generated code, the majority of the loads and stores access one word at a time. We denote the time to load or to store a single noncacheable word by $T_{noncacheable}$.

With these assumptions, we can simplify the memory access time in Formula (1) to

$$\begin{aligned} T = & T_{miss} * (N_{main} + N_{mini} + N_{noncacheable}) \\ & -(T_{miss} - T_{hit}) * (N_{main} * h_{main} + N_{mini} * h_{mini}) \\ & -(T_{miss} - T_{noncacheable}) * N_{noncacheable} \end{aligned} \quad (2)$$

Notice that, under the condition of $T_{noncacheable} = T_{miss}$, T is minimized if and only if the total number of hits, i.e. $N_{main} * h_{main} + N_{mini} * h_{mini}$, is maximized. In the following, we shall first prove the problem of maximizing the cache hits to be NP-hard. We then prove the NP-hardness of the CACHE-MAPPING problem without the condition of $T_{noncacheable} = T_{miss}$.

3.1 NP-hardness proof

Definition 1. MAX-HIT problem:
Instance: A main cache of size S, a mini-cache of size S_{mini}, each having either the LRU or the Round-Robin replacement policy, a page size S_p, a set ($\mathcal{P}$) of virtual pages such that each page $P_i \in \mathcal{P}$ contains memory blocks $(i,1),(i,2),\ldots,(i,S_p)$, and a sequence of memory accesses $A = a_1,\ldots,a_n$, to the memory blocks introduced above. The number of distinct memory blocks accessed in the sequence is assumed to be greater than the size of each cache.
Solution: a partition of pages in $\mathcal{P}$ into Set_{main}, Set_{mini} and $Set_{noncacheable}$, such that the total number of cache hits of A is maximized.

Lemma 1: MAX-HIT is NP-hard in terms of the length of the memory-access sequence if $max(S, S_{mini}) \leq S_p - 1$ and $S \neq S_{mini}$.
Proof: We reduce MAX2SAT [8] to MAX-HIT. The MAX2SAT problem is defined as: given a set of clauses, each being a disjunction of at most two literals[1], and an integer K, whether there is a truth assignment that satisfies at least K of the clauses. Given an instance of MAX2SAT, we construct a sequence of memory accesses which consists of a prefix and a postfix. The prefix enforces a one-to-one correspondence between the truth assignment in the MAX2SAT and the page placement in MAX-HIT. The postfix is transformed from the clauses of the given MAX2SAT instance.

Throughout this proof, we use the notation $A(i,b)$ to represent a reference to the b^{th} memory block in page P_i (if $b = 0$, $A(i,b)$ is null). The notation $A[(i,b_1),(i,b_2)]$ denotes the series of $A(i,b_1), A(i,b_1-1),\ldots,A(i,b_2)$. If $b_1 < b_2$, $A[(i,b_1),(i,b_2)]$ is empty. Let N be the length (i.e. the number of clauses) of the MAX2SAT instance. Without loss of generality, we assume $S > S_{mini}$.

We first construct the prefix. For each variable v in MAX2SAT, we introduce 3 virtual pages, P_v, $P_{\neg v}$ and $P_{v''}$. The prefix is the concatenation of the following memory accesses for each variables v:
$A[(v,S),(v,1)]A[(\neg v,S),(\neg v,1)]A(v'',1)$ ...(repeat $2*N+1$ more times)
By placing $P_{v''}$ in Set_{mini} (i.e. mapping it to the mini-cache), either P_v or $P_{v'}$ in Set_{main} (i.e. mapping it to the main cache) and the other page in $Set_{noncacheable}$, the prefix has $(2*N+1)*(S+1)$ cache hits,the maximum possible.

Table 1 defines the rules to transform a clause to the memory accesses in the postfix. In the table, α and β denote two literals of distinct variables in the clause[2]. α' and β' are the opposite literals respectively. For each clause in the MAX2SAT instance, we apply one of the two rules exactly once. Since $S \leq S_p - 1$, we have at least $S+1$ memory blocks in each virtual page. Given the cache sizes, the memory accesses introduced for each clause have 2 potential cache hits (marked in **bold**). Therefore, the total number of cache hits in the postfix is at most $2*N$.

We obtain the whole memory access sequence by appending the postfix to the prefix. In the optimal solution, exactly one of the two pages associated with each variable should be mapped to the main cache:

- Mapping P_v or $P_{v'}$ to the mini-cache will not produce any hit in the mini-cache for the memory accesses in prefix. As a result, only page $P_{v''}$ should be mapped to the mini-cache. Otherwise, we lose at least $(2*N+1)$ hits.

- If there exists a variable v such that both P_v and $P_{\neg v}$ are in Set_{main}, the accesses to P_v and $P_{\neg v}$ in the prefix will be cache misses. By placing one of them in $Set_{noncacheable}$, the cache hit count will increase by at least $(2*N+1)*S$. In the postfix, while this

[1] v and $\neg v$ are two opposite literals of variable v.

[2] $\alpha \vee \alpha$ is reduced to α, and $\alpha \vee \neg\alpha$ is removed.

Table 1. Clause transformation rules under $S \leq S_p - 1$

Clause	Memory access sequence
α	$A[(\alpha, S+1),(\alpha,1)]\ \mathbf{A}(\alpha,\mathbf{1})$
$\alpha \vee \beta$	$A[(\alpha, S+1),(\alpha,1)]\ \mathbf{A}(\alpha,\mathbf{1})\ \ A[(\alpha', S+1),(\alpha',2)]\ A(\beta', S+1)\ A(\alpha', 1) A[(\beta', S),(\beta',1)]\ \mathbf{A}(\alpha',\mathbf{1})$

may force some accesses to become cache misses, the decrease in cache hits is no more than $2 * N$.

- By the same argument, if there exists a variable v such that both P_v and $P_{\neg v}$ are in $Set_{noncacheable}$, by placing one of them in Set_{main}, the total cache hit count will be increased.

Therefore, we can build the following one-to-one mapping between the truth assignment and the page placement:

- $P_v \in Set_{main} \Leftrightarrow v$ is *true*.
- $P_{\neg v} \in Set_{main} \Leftrightarrow v$ is *false*.

Under this mapping, the transformation rules guarantee that a satisfied clause will increase the cache hit count by exactly 1, and an unsatisfied clause will not affect the cache hit count:

- If α is *true* (which means P_α is in Set_{main}), the memory access $\mathbf{A}(\alpha, \mathbf{1})$ is a hit, but no other listed accesses can be hits.
- If α is *false* (which means $P_{\alpha'}$ is in Set_{main}), the memory access $\mathbf{A}(\alpha, \mathbf{1})$ will not be a cache hit.
  - If β is *true*, $P_{\beta'}$ is in $Set_{noncacheable}$, so $\mathbf{A}(\alpha', \mathbf{1})$ is a hit.
  - If β is *false*, $P_{\beta'}$ is in Set_{main}, $\mathbf{A}(\alpha', \mathbf{1})$ is a miss. The cache hit count will not increase.

With this property, it is easy to see that maximizing the cache hit count is equivalent to maximizing the number of satisfied clauses. An optimal MAX-HIT solution will derive an optimal solution of the MAX2SAT.

Finally, the memory access length constructed in this reduction is a polynomial function of N. Therefore, MAX-HIT is NP-hard. $\diamond$

Note that in the proof given above, we do not assume any associativity property for the caches. The proof is valid for directly-mapped, set-associative or full-associative caches.

Lemma 2: Lemma 1 remains correct if $S = S_{mini}$.
Proof: We will use the following prefix,
$A[(v,S),(v,1)]A[(\neg v,S),(\neg v,1)]A(v'',1)A[(v'',S),(v'',1)]$
$\ldots$(repeat $2 * N + 1$ more times)
In the optimal mapping, $P_{v''}$ will definitely be mapped to one of the two caches. Exactly one of P_v and $P_{\neg v}$ will be mapped to the other cache. If P_v is mapped to the cache, v is *true*, otherwise, v is *false*. $\diamond$

Lemma 3: If $max(S, S_{mini}) \geq S_p$ and the caches are fully associative, MAX-HIT is still NP-hard in terms of the length of the memory access sequence.
Proof: See [17]. $\diamond$

Theorem 1: CACHE-MAPPING is NP-hard in terms of the length of the memory access sequence.
Proof: Suppose $T_{miss} \neq T_{noncacheable}$. The optimal cache mappings given in the proofs of Lemmas 1 through 3 remain optimal, as long as we make the prefix sufficiently long. We show how this is done in Lemma 1. In the prefix, we repeat memory accesses to P_v and $P_{\neg v}$ R more times, instead of $2 * N + 1$ more times, where R is determined as follows. Suppose we change the page mapping from the optimal one in Lemma 1. This would increase T in the prefix by at least $R * (T_{noncacheable} - T_{hit})$. At the same time, we may decrease T in the postfix by no more than $2(S+1) * N * (T_{miss} - T_{noncacheable})$. Let $R = 2(S+1) * N * (T_{miss} - T_{noncacheable})/(T_{noncacheable} - T_{hit}) + 1$. The optimal mapping in Lemma 1 will remain optimal for CACHE-MAPPING. The number of satisfied clauses in MAX2SAT equals the number of cache hits in the optimal solution for CACHE-MAPPING. $\diamond$

If we remove the mini-cache from CACHE-MAPPING, the problem becomes how to optimally select the non-cacheable virtual pages. With slight adjustments, the proofs of Lemmas 1 through 3 and Theorem 1 remains valid. (In the proof of Lemma 1, we simply remove the memory accesses to $A(v'', 1)$ from the prefix.) Hence, this special case of CACHE-MAPPING is also NP-hard.

4 A Heuristic Algorithm

Because the CACHE-MAPPING problem is NP-hard, we develop a heuristic algorithm to obtain an approximate solution in polynomial time.

To better illustrate the idea of the heuristic, we use the following code example throughout this section. We

assume that the arrays in the example are allocated contiguously in the memory, the start address of array A is page aligned and the cache parameters are the same as StrongARM SA-1110 (refer to Section 2).

```
int A[1024]; /*  Page P1 */
int B[24];   /*  Page P2 */
int C[1000];
int D[2300]; /* Page P3 P4 and P5 */
int E[1024]; /* Page P5 and P6 */
        ...
        for(i=0;i<20;i++) {
            for(j=0;j<1024;j++)
                A[j] += i+j + B[j%24];
            for(j=0;j<1024;j++)
                A[j]+= D[j] +C[j%512];
        }
        for(i=1024;i<2048;i++)
             E[i-1024] = D[i];
```

The heuristic uses a greedy strategy to try to fit the most accessed pages into the caches. We visit each virtual page in the decreasing order of their access counts. In the example code, the page visit order is P_1 P_2 P_3 P_4 P_5 P_6. We start with the system's default mapping which maps all pages to the main cache. We check to see whether we will move the visited page from the main cache to the mini-cache or mark it as noncacheable. Initially, all pages are marked as *undecided*. Once we mark a page as *decided*, its mapping will not change any more.

When we pick the most accessed *undecided* page P_i for mapping, we count the number of cache hits after taking one of the following two actions:

1. Map P_i to the mini-cache.
2. Keep P_i in the main cache and map some other *undecided* pages to the mini-cache. These candidates for mapping to the mini-cache are picked according to their *conflict weights* against page P_i, which will be explained later.

To count the cache hits, we assume the caches to be fully associative and the replacement policy to be LRU. For each memory reference, r, which accesses memory block a, we count Rd, the number of distinct memory blocks which are mapped to the same cache as a (under the current mapping) and which are accessed between r and the most recent reference to a. If Rd is smaller than the size of the cache to which a is mapped, then r is a hit. Otherwise r is a miss. To make computation of Rd fast, we preprocess the memory trace by keeping a *page-population list* for each reference r. Each member in such a list is a pair (k, c_k), where k represents a virtual page referenced between r and its most recent reference to the same memory address a. c_k is the number of distinct memory blocks in page k which are referenced meanwhile. To compute Rd for r, under any given page mapping, we just need to add up the reference counts of the pages (in the page-population list) which are mapped to the same cache as the page containing r. This takes $O(m)$ time for each reference r, where m is the number of pages. The preprocessing has a one-time cost of $O(nm)$, where n is the total number of the memory references.

For each choice (Action 1 or Action 2) described above, we not only count the total cache hits as the result, but also the cache hit ratio for P_i and those candidate pages for moving to the mini-cache. We favor the choice which results in a higher total number of hits. However, before we finally commit to the page mapping according to that choice, we examine the hit ratio for each of the pages to be decided, to see whether it is below the *noncacheable threshold* (which makes the page noncacheable). If the hit ratio, h, for page p satisfies

$$h < \frac{T_{miss} - T_{noncacheable}}{T_{miss} - T_{hit}}$$

then we have

$$T_{hit} * N(p) * h + T_{miss} * N(p) * (1-h) > T_{noncacheable} * N(p)$$

where $N(p)$ is the number of memory access to page p. Hence, p should be made noncacheable. If we determine that a candidate page stays mapped to a cache, we commit its recent mapping decision and mark it as *decided*. For our experimentation platform, i.e. the Intel StrongARM SA-1110, we compute the *noncacheable threshold* to be 0.28.

For Action 2 mentioned above, we pick a set of *undecided* pages as candidates for mapping to the mini-cache. This set is selected based on the *conflict weights* between P_i and those candidate pages.

Let $a_{i,j}$ be the j^{th} access in page i and let $r_{i,j}$ be the most recent reference to the same memory block accessed by $a_{i,j}$. $Rd(a_{i,j})$ is the number of distinct memory blocks accessed between $a_{i,j}$ and $r_{i,j}$. The *conflict weight* between page P_i and page P_k is defined as $W_i(P_k) \equiv \sum_j \frac{R1(k,a_{i,j})}{Rd(a_{i,j})}$, where $R1(k, a_{i,j})$ is the number of distinct memory blocks in page P_k that are referenced between $a_{i,j}$ and $r_{i,j}$.

We rank all currently *undecided* pages by the decreasing order of their *conflict weights* against P_i, and we try one by one to select those to be mapped to the mini-cache. In the sample code, when visiting page P_1, we compute $W_1(P_2) = 990$, $W_1(P_3) = 1187$ and the *conflict weights* of other pages against P_1 equal 0. Hence, we will first map P_3 to the mini-cache. If this move increases the overall cache hits, then in the next step, we will try to map both P_3 and P_2 to the mini-cache. Otherwise, we will pull back P_3 and try to map P_2 to the mini-cache. The decision made when we visit P_1 is to map P_1 to the main cache and P_3 to the mini-cache. Their hit ratios are both above the noncacheable threshold.

Our final mapping result is P_1 and P_2 to the main cache and all the remaining pages to the mini cache. We reduce the cache misses from 6862 (by the default mapping) to 3010.

The overall algorithm is presented in Figure 1. The input of the heuristic is the memory trace. The complexity of the algorithm is $O(m^3 n)$ where m is the number of pages and n is the number of memory accesses. Since, in each step, the mapping changes only if the total memory access time decreases, it is easy to see that our heuristic algorithm will never generate a cache mapping worse than the default mapping where all pages mapped to the main data cache.

```
Procedure Greedy_cache_mapping
Input: memory trace
Output: Set_main, Set_mini and Set_noncacheable
Procedure:
  Initialize Set_mini = Set_noncacheable = ∅
  Initialize Set_main = all pages; Mark all pages undecided
  Sort pages by the page access count
  For each undecided page P_i picked in decreasing order of the access count
    Hits = the total hit count on both caches
    Try putting P_i in Set_mini
    Compute try1 = the new total hit count on both caches
    Mark page P_i as candidate
    If (Hits < try1)
      Hits = try1
    Endif
    Compute the conflict weights for Page P_i
    For each undecided page P_j picked in decreasing order of W_i(P_j)
      Try putting P_j in Set_mini
      Compute try2 = the new hit count on both caches
      If (Hits < try2)
        Put P_j in Set_mini
        Hits = try2
        Mark page P_j as candidate
      Endif
    Endfor
    If (Hits = try1)
      Put P_i in Set_mini
    Endif
    If (Hit ratio of a candidate page P_j < noncacheable threshold)
      Move P_j to Set_noncacheable
    Endif
    Mark candidate page(s) as decided
  Endfor
End
```

Figure 1. The heuristic algorithm

5 Experimental Results

We have implemented the cache mapping heuristic for the Compaq iPAQ 3650 PDA, which has a 206MHz Intel StrongARM SA-1110 processor and 32MB RAM. Since changing the default mapping requires the modification of the page table entry, we add a system call to the Linux kernel on the iPAQ. This system call modifies the page table entry for one virtual page and also maintains the consistency of the caches. We insert a number of such system calls in the original program to enforce the new cache mapping determined by the heuristic. Notice that the system call is inserted just once for each page that has a non-default mapping. The run time overhead of executing the instrumented code is less than 0.8% of the total execution time for all the test programs. The memory traces of the test programs are generated by a SimpleScalar [3] based simulator. Throughout the experiment, we use `gcc` with the `-O3` switch to compile the program.

The programs used in the experiments are: `pegwit/decryption`, `pegwit/encryption`, `ADPCM/rawcaudio`, and `ADPCM/rawdaudio` from MediaBench, `mcf`, `crafty` and `gzip` from SPEC2000 benchmark (the test input size), `mm`, a matrix-multiply program (the problem size is 397), and `ncompress`, a UNIX compression utility. We choose these programs because they have relatively poor cache performance in the iPAQ for the default mapping under `gcc`, where all the virtual pages are mapped to the main data cache. For programs (in MediaBench and SPEC2000) that have good cache performance, the heuristic selects non-default mapping for only a few rarely accessed pages. There is no noticeable improvement or degradation for the new cache mapping for these programs.

Figure 2(a) shows the measured execution time of test programs after applying the new cache mapping. Since the running time of different programs varies significantly, we normalize it to the original program. The performance improvements range from 1% to 21%. The harmonic mean of the speedup is 1.12.

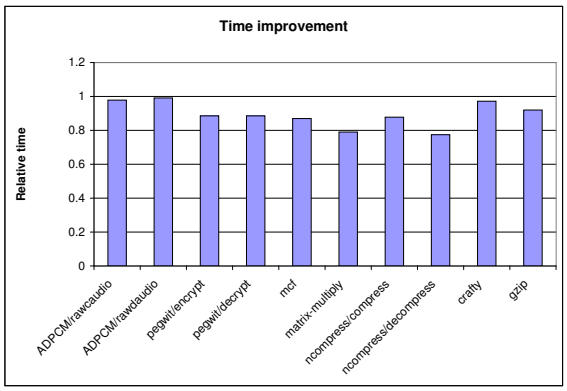
For a handheld device, it is important to find out whether any performance gain comes at the cost of increased energy consumption (due to the use of additional hardware units, for example). In order to find out whether the new cache mapping increases energy consumption, we measure the energy saving in the new cache mapping. The experimental settings for energy measurement are the same as described in [16]. Figure 2(b) plots the energy consumed by each program. The energy consumed by the original program is used as the base for normalization. We see that the new cache mapping reduces the energy consumption of the handheld device by 4% to 28% for the listed test programs, with a arithmetic mean of 16%.

As discussed in Section 4, the complexity of our heuristic is relatively high. A simpler heuristic can be randomly mapping virtual pages between the mini-cache and the main cache in proportion to their sizes, and using the *noncacheable threshold* to mark pages noncacheable. Real measurements show that this random mapping increases execution time over the default mapping by 6% to 180% for the listed test programs.

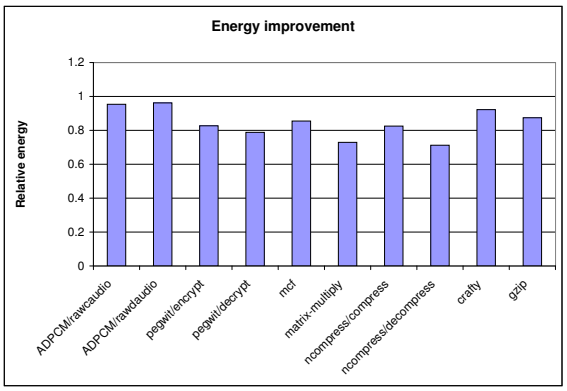
5.1 Memory performance

We modify the SimpleScalar simulator to collect detailed information on memory performance.

Table 2 presents the memory system statistics for the default cache mapping with all pages mapped to the main data cache. The column with the label of *ld/st #* shows the number of loads and stores of the program. In the following column we present the memory performance for a 8KB 32-way set-associative cache with the Round-Robin replacement policy (which is the configuration of the main data cache in StrongARM SA-1110). The *cache hit #* subcolumn shows the number of cache hits. The next two subcolumns, *main memory access #* and *main memory data amount*, show the total number of memory accesses and total amount of data traffic to the main memory. The column with label *16KB data cache* shows similar information but for a larger cache (a 16KB 32-way set-associative cache



(a) Execution time



(b) Energy

Figure 2. Normalized execution time and normalized energy consumption on the handheld device

Table 2. Memory system statistics (when only the main data cache is used)

		8KB data cache			16KB data cache		
program	ld/st # ($\times$1000)	cache hit # ($\times$1000)	main memory access # ($\times$1000)	main memory data amount (MB)	cache hit # ($\times$1000)	main memory access # ($\times$1000)	main memory data amount (MB)
ADPCM/rawcaudio	40385	37356	3107	99.4	37978	2456	78.6
ADPCM/rawdaudio	50826	45744	5159	165.1	45947	4896	156.7
pegwit/encrypt	13024	10914	2160	69.1	11075	1978	63.4
pegwit/decrypt	7457	6204	1281	41.0	6293	1182	37.8
mcf	59630	45855	14866	475.7	48175	12448	398.3
mm	313491	242311	71181	2277.6	289397	24094	771.1
ncompress/compress	9791	5590	4277	136.9	5816	4050	129.6
ncompress/decompress	8261	5601	2698	86.3	6130	2149	68.8
crafty	19842	18235	1722	55.1	18718	1193	38.2
gzip	28893	23548	5573	178.3	24328	4755	152.1

Table 3. Memory system statistics (when the cache mapping determined by the heuristic is applied)

	main cache		mini-cache		noncacheable		cache total		main memory	
program	access # ($\times$1000)	hit # ($\times$1000)	access # ($\times$1000)	hit # ($\times$1000)	access # ($\times$1000)	data amount (MB)	cache access # ($\times$1000)	hit # ($\times$1000)	access # ($\times$1000)	data amount (MB)
rawcaudio	38181	36539	2213	1936	0	0	40393	38475	1908	61.1
rawdaudio	47516	44380	3319	2659	0	0	50835	47039	3747	120.0
encrypt	12210	10952	33	18	781	3.1	12243	10970	2080	44.7
decrypt	7032	6224	17	9	407	1.6	7050	6233	1240	28.3
mcf	30794	28300	28785	20265	53	0.2	59580	48565	11871	378.4
mm	126741	126384	124554	124515	62199	248.8	251295	250899	62662	263.6
compress	6087	5375	624	455	3080	12.3	6711	5830	4011	42.1
decompress	5939	5458	545	441	1778	7.1	6484	5898	2383	26.4
crafty	17055	15674	2690	2655	97	0.2	19745	18329	1616	48.8
gzip	22989	21449	3839	2161	2066	8.3	26828	23610	5372	114.1

Table 4. The reduction of the amount of data accessed at the main memory

program	total	factor (2)	factor (1)	program	total	factor (2)	factor (1)
rawcaudio	38.5%	3.3%	35.2%	mm	88.8%	77.5%	11.3%
rawdaudio	27.3%	2.8%	24.5%	compress	69.2%	63.8%	5.4%
encrypt	35.3%	32.8%	2.5%	decompress	69.4%	58.7%	10.7%
decrypt	31.0%	28.8%	2.2%	crafty	11.4%	6.1%	5.3%
mcf	20.5%	2.7%	17.8%	gzip	35.9%	34.8%	1.1%

also with Round-Robin policy).

Table 3 presents the memory access statistics for the new cache mapping. The parameters for the caches are the same as the StrongARM SA-1110. The *main cache* and *mini-cache* columns show the breakdown of the accesses for two caches and the corresponding numbers of cache hits. The *cache total* column is the combined cache access count and number of cache hits for cacheable pages. The *noncacheable* and *main memory* columns show the memory access count and amount of accessed data for noncacheable pages and for the main memory, respectively. Note that since we have instrumented codes to enforce the new cache mapping, the total number of accesses is slightly greater than the number of loads and stores in Table 2.

Comparing Table 3 with Table 2, we see that the new cache mapping can enhance the overall memory performance through two factors: (1) increasing the overall cache hits, and (2) saving the data amount by avoiding fetching unused data. Both cache bypass and the use of the mini-cache can contribute to factor (1). For example, in `rawcaudio`, the heuristic places the input and output buffers, which exhibit spatial locality only, in the mini-cache. This way, we do not lose cache hits for the accesses to the virtual pages containing the input/output buffers, and the cache hits to other pages increase. Program `mm` shows an example of using cache bypass to increase overall hits. This program computes the matrix multiplication of C = A * B. The heuristic places the pages for array C into the mini-cache, marks the pages for array B as noncacheable, and keeps the pages for array A in the main cache. By doing this, we increase the number of hits through the temporal reuse for the references in the pages of array A, and the data transfer amount to the main memory is also significantly reduced (by bypassing array B). It is worth while to mentioning that the new cache mapping determined by the heuristic increases the overall cache hits for all the test program.

The increase of the overall hits also reduces the data amount fetched from the main memory. However, if cache bypass is heavily applied, factor (2) tends to be the bigger component in the overall data amount reduction. Table 4 shows the composition of the reduction. The *total* column shows how much the new mapping reduces the amount of data accessed at the main memory, which is normalized to data amount of the original program. The next two columns show such reductions from factor (2) and factor (1), respectively. We see that for `encrypt`, `decrypt`, `mm`, `compress`, `decompress`, and `gzip`, a predominant percentage of the reduction is from factor (2). This shows that these programs benefit more from cache bypass, since factor (2) is solely achieved from cache bypass.

We also perform experiments to understand whether the improvements of hit ratios are the result of a relatively larger cache (i.e. 8K+512 bytes), or the result of smart use of the cache mapping method. We can compare Table 3 with the data of *16KB data cache* column in Table 2. Although with a much larger-sized cache (16KB), the hit ratios for `rawcaudio`, `rawdaudio` and `mcf` are still lower than achieved by the new mapping on the smaller cache configuration of (8K + 512 bytes).

This indicates the improvements are indeed from the new cache mapping.

6 Related Work

Cache bypass and the horizontally partitioned caches have been studied in [23, 13, 9, 20, 21, 22, 18]. Most of these previous efforts focus on new cache designs at the micro-architectural level. Our work in this paper studies a software method to better utilize the caches on a commercial system that are horizontally partitioned and allow cache bypass. We have not found previous work that addresses the same issue. To the best of our knowledge, this is the first work to model the cache mapping problem at virtual page level and report measurement result (both for performance and for energy data).

Several authors have applied heuristics to memory traces to determine memory allocation for data objects [4, 7, 6]. They rearrange the layout of data objects to reduce conflict misses. (Optimal data layout is known to be NP-hard [19].) Our scheme does not change the data layout and is aimed at reducing capacity misses by smart page mapping.

Brehob et al. [2] prove that the optimal replacement for *Companion Cache Structure (CCS)* is NP-hard. A CCS consists of a main cache and a fully-associative companion buffer (similar to the mini-cache). Cache bypass is allowed for both caches in CCS. Their work is different from this work in the following major aspects. First, they assume ideal cache replacement, while we assume LRU or Round-Robin replacement. Second, their proofs require the companion buffer. In contrast, our proofs are always valid, regardless the existence of the mini-cache. Third, the cache mapping decision in our work is for each virtual page. Their work can be viewed as a special case of the page size equaling to one memory block. For other page sizes, their proof is not usable.

In [12], Johnson and Hwu present polynomial-time algorithms to obtain an upper bound of the hit ratio improvements with cache bypassing. They assume, however, that the system allows the bypass decision to be made differently for different references to the same memory address. This is in contrast to the bypass mechanism on the StrongARM SA-1110 where the bypass decision is statically made for each virtual page. Hence, their reference marking method cannot be used.

7 Conclusions

In this paper, we have presented a model of the cache mapping problem and prove that the optimal cache mapping is NP-hard. A heuristic is given to select the cache mapping. We implement the cache mapping heuristic for Compaq iPAQ 3650 handheld devices running the Linux operation system. Execution time measurement shows performance enhancement up to 21% for a set of test programs. As a byproduct of performance enhancement, we also save the energy up to 28%.

Acknowledgments

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

References

[1] D. H. Albonesi. Selective cache ways: On-demand cache resource allocation. In *Proceedings of the 32nd International Symposium on Microarchitecture*, pages 248–259, 1999.

[2] M. Brehob, S. K. Wagner, E. Torng, and R. J. Enbody. Optimal replacement is np-hard for nonstandard caches. *IEEE Transactions on Computers*, 53(1):73–76, 2004.

[3] D. Burger and T. Austin. The SimpleScalar tool set, version 2.0. Technical Report 1342, University of Wisconsin-Madison, June 1997.

[4] B. Calder, K. Chandra, S. John, and T. Austin. Cache-conscious data placement. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, San Jose, 1998.

[5] K. K. Chan, C. C. Hay, J. R. Keller, G. P. Kurpanek, F. X. Schumacher, and J. Zheng. Design of HP PA 7200 CPU. In *Hewlett-Packard Journal*, February 1996.

[6] T. M. Chilimbi, B. Davidson, and J. R. Larus. Cache-conscious structure definition. In *Proceedings of SIGPLAN'99 Conference on Programming Languages Design and Implementation*, pages 13–24, Atlanta, Georgia, 1999.

[7] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Cache-conscious structure layout. In *Proceedings of SIGPLAN'99 Conference on Programming Languages Design and Implementation*, pages 1–12, 1999.

[8] M. R. Garey, D. S. Johnson, and L. J. Stockmeyer. Some simplified NP-complete graph problems. *Theoretical Computer Science*, 1:237–267, 1976.

[9] A. Gonzalez, C. Aliagas, and M. 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.

[10] Intel Corporation. Intel StrongARM SA-1110 microprocessor developer's manual. http://www.intel.com/design/strong/manuals/278240.htm, October 2001.

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

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

[13] T. L. Johnson and W. 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.

[14] J. Kin, M. Gupta, and W. H. Mangione-Smith. The filter cache: An energy efficient memory structure. In *International Symposium on Microarchitecture*, pages 184–193, 1997.

[15] H. S. Lee and G. S. Tyson. Region-based caching: an energy-delay efficient memory architecture for embedded processors. In *Proceedings of the international conference on Compilers, architectures, and synthesis for embedded systems*, pages 120–127. ACM Press, 2000.

[16] Z. Li, C. Wang, and R. Xu. Computation offloading to save energy on handheld devices: A partition scheme. In *International Conference on Compiler, Architecture and Synthesis for Embedded Systems*, pages 238–246, Atlanta, Georgia, November 2001.

[17] Z. Li and R. Xu. Np-hardness of cache mapping. Computer Sciences Technical Report TR 04-001, Purdue University, 2004. Also available at http://www.cs.purdue.edu/homes/xur/research/cm_tech.pdf.

[18] 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.

[19] E. Petrank and D. Rawitz. The hardness of cache conscious data placement. In *Proceedings of the 29th Annual ACM Symposium on Principles of Programming Languages*, pages 101 – 112, Portland, Oregon, January 2002.

[20] J. A. Rivers and E. 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.

[21] J. A. Rivers, E. S. Tam, G. S. Tyson, E. S. Davidson, and M. Farrens. Utilizing reuse information in data cache management. In *Conference proceedings of the 1998 international conference on Supercomputing*, pages 449–456. ACM Press, 1998.

[22] M. Tomasko, S. Hadjiyiannis, and W. A. Najjar. Evaluation of a split scalar/array cache architecture. In *IEEE Computer Society Technical Committee on Computer Architecture: Special Issue on Distributed Shared Memory and Related Issues*, pages 11–16, March 1997.

[23] G. Tyson, M. Farrens, J. Matthews, and A. 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.

[24] O. S. Unsal, I. Koren, C. M. Krishna, and C. A. Moritz. The minimax cache: An energy-efficient framework for media processors. In *HPCA*, pages 131–140, 2002.