2004

NP-Hardness of Cache Mapping

Zhiyuan Li
Purdue University, li@cs.purdue.edu

Rong Xu

Report Number:
04-001

NP-HARDNESS OF CACHE MAPPING

Zhiyuan Li
Rong Xu

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

CSD TR #04-001
January 2004

NP-hardness of Cache Mapping*

Zhiyuan Li Rong Xu †
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907
{li,xur}@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 model the cache mapping problem and prove that finding the optimal cache mapping is NP-hard.

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 [9, 8, 11, 2] 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 [8, 11, 1, 7], 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 a 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 [6, 10] have been proposed to monitor the memory access patterns and to make bypass decisions.

Processors such as the Intel StrongARM SA-1110 [4] and the Intel XScale [5] allow 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 [4] 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: (1) It is often difficult to predict the cache reuse pattern in non-numerical programs. Even the programmer may not predict the cache behavior accurately. (2) 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.

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 caches with cache mapping

*Technique Report CSD-TR-04-001, Department of Computer Sciences, Purdue University, West Lafayette, IN 47907, January, 2004

†The author names are listed in alphabetical order.

can perform better than traditional caches. In Section 3 we model the cache mapping problem and prove it to be NP-hard.

2 Cache Mapping Problem

The Intel StrongARM SA-1110 processor [4] 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. Although under certain circumstances, an optimizing compiler may be able to use software means to influence the replacement policy (memory overlay and reference reordering are two well-known techniques of such), unfortunately, dependence relations in a program often prevent compilers to reorder memory references. Opportunities for memory overlay are also limited because many variables may remain live at the same time. Independently indexed caches offer opportunities to manipulate the replacement policy without being constrained by dependence relations or live variable information.

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 have 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:

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

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 i^{th} noncacheable memory access size. 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 dominate 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 because although the memory

system supports accesses in the *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 $T =$

$$T_{miss} * (N_{main} + N_{mini} + N_{noncacheable}) - \\ (T_{miss} - T_{hit}) * (N_{main} * h_{main} + N_{mini} * h_{mini}) - \\ (T_{miss} - T_{noncacheable}) * N_{noncacheable} \quad (2)$$

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. CACHE-MAPPING 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 memory access time T defined in Equation 2 is minimized.

Definition 2. 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 [3] 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 accesses sequence by appending the postfix to the prefix. In the optimal solution, exact 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

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

prefix are 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 may force some accesses to become cache misses, the decreases 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 trace length constructed in this reduction is a polynomial function of N. Therefore, MAX-HIT is NP-hard. ◇

Note that in the proof, 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,
$A[(v, S), (v, 1)] A[(\neg v, S), (\neg v, 1)] A(v'', 1) A[(v'', S), (v'', 1)]$
... (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*. ◇

Lemma 3: If $max(S, S_{mini}) \geq S_p$, MAX-HIT is still NP-hard in terms of the length of the memory accesses sequence for fully-associative caches.
Proof: We still reduce MAX2SAT to MAX-HIT, using the same notations in Lemma 1. We assign 2 pages, P_v and $P_{\neg v}$, for each variable v, in the MAX2SAT instance.

Let $m = \lfloor S/S_p \rfloor$, $S_{new} = S \% S_p$, $m1 = \lfloor S_{mini}/S_p \rfloor$, and $S1_{new} = S_{mini} \% S_p$. We introduce $m + m1 + 1$ padding pages, numbered from 1 to $m + m1 + 1$.

Let $Seq_{padding}$ represent the following memory access sequence to the padding pages, $Seq_{padding} = A[(1, S_p), (1,1)]$ ... $A[(m-1, S_p), (m\text{-}1,1)]$ $A[(m, S_{new}), (m,1)]$ $A[(m+1, S_p), (m\text{+}1,1)]$ ... $A[(m\text{+}m1, S_p), (m\text{+}m1,1)]$ $A[(m\text{+}m1\text{+}1, S1_{new}), (m\text{+}m1\text{+}1,1)]$. $Seq_{padding}$ accesses $S + S_{mini} - S_p$ memory blocks. If $S_{new} = 0$, then page P_m does not exist. If $S1_{mini} = 0$, then page P_{m+m1+1} does not exist. Accesses to such nonexistent pages should be removed throughout the proof.

For each clause in the MAX2SAT instance, we use one of the two rules in Table 2 exactly once to generate a sequence of memory access in the postfix. Each clause contains at most $2*(S+S_{mini})+S_p+4$ memory accesses. So the total number of cache hits for all the clauses does not exceed $(2*(S+S_{mini})+S_p+4)*N$.

Let $Seq\_lit(v, R)$ be R repetitions of $Seq_{padding}$ $Seq_{padding}$ $A[(v, S_p), (v,1)]$ $A[(\neg v, S_p), (\neg v,1)]$. The prefix contains the following memory access sequence:
$Seq\_lit(v_1, R)\ Seq\_lit(v_2, R)\ \ldots\ Seq\_lit(v_{N_v}, R)$
where N_v is the number of variables in the MAX2SAT instance.

We first examine what kind of cache mapping maximizes the number of cache hits in $Seq\_lit(v, R)$ for each v, where $R > 1$.

- All memory blocks which are accessed in $Seq\_lit(v, R)$ and are not placed in $Set_{noncacheable}$ should exactly cover both caches. Hence, if $S_{new} > 0$, then P_m should not be in $Set_{noncacheable}$. Neither should

Table 2. Clause transformation rules under $S \geq S_p$

Clause	Memory accesses sequence
α	$Seq_{padding}\ A(m, S_{new}+1)A(m+m1+1, S1_{new}+1)A[(\alpha, S_p), (\alpha, 1)]\ \mathbf{A}(\alpha, \mathbf{1})$
$\alpha \vee \beta$	$Seq_{padding}\ A(m, S_{new}+1)A(m+m1+1, S1_{new}+1)A[(\alpha, S_p), (\alpha, 1)]\ \mathbf{A}(\alpha, \mathbf{1})$ $A[(\alpha', S_p), (\alpha', 1)]\ Seq_{padding}\ A[(\beta', S_p), (\beta', 1)]\ \mathbf{A}(\alpha', \mathbf{1})$

P_{m+m1+1} if $S1_{new} > 0$. Further, if $S1_{new} \neq S_{new}$, then P_m should be mapped to the main cache and P_{m+m1+1} to the mini-cache. If $S1_{new} = S_{new}$, then either page can be mapped to the main cache, the other to the mini-cache.

- For each variable v, exactly one of P_v and $P_{\neg v}$ should be mapped to either the main cache or the mini-cache. If both were in $Set_{noncacheable}$, then we would leave S_p cache lines unused. On the other hand, if both P_v and $P_{\neg v}$ were mapped to the caches, then one of the padding pages we introduced would be in $Set_{noncacheable}$. Since each padding page is accessed more frequently than either of P_v and $P_{\neg v}$, placing either P_v or $P_{\neg v}$ in $Set_{noncacheable}$ will result in more cache hits.

For any of the variables, if either of the two requirements mentioned above is unsatisfied, then we would lose at least $R - 1$ cache hits in the prefix. Since the postfix can have no more than $2 * (S + S_{mini}) + S_p + 4) * N$ cache hits, we simply let $R = 2 * (S + S_{mini}) + S_p + 4) * N + 2$ to make sure that any optimal mapping will satisfy the two requirements above for all variables.

It is important to note that all those pages which are assigned to the variables and not in $Set_{noncacheable}$ will be mapped to the same cache (i.e. either all in Set_{main} or all in Set_{mini}). Thus, since both caches are fully associative, the memory access sequence corresponding to each satisfied clause (see Table 2) will have exactly one cache hit, and an unsatisfied clause will generate no cache hits. Maximizing the number of satisfied clauses is equivalent to maximizing the number of cache hits in the entire memory access sequence. The MAX-HIT problem is NP-hard. ◇

Lemma 4: For virtual-indexed caches, Lemma 3 remains correct if the cache is set-associative or direct-mapped.
Proof: If $S \geq S_p$ and the cache is virtual-indexed, the proof of Lemma 2 can be modified to work for the set-associative or direct-mapped cache. We manipulate the virtual page numbers to make sure that every page created for the variables is mapped to the lowest S_p cache lines. We also make sure that P_1 through P_m cover cache-line indices from $S_p + 1$ to S. P_{m+1} through P_{m+m1+1} cover cache-line indices from 1 to S_{mini}. Notice that, for convenience, we write the lowest cache index 1 (instead of 0 by convention). These treatments will ensure that, by the rules in Table 2, a satisfied clause will generate exactly one cache hit and an unsatisfied clause generate no cache hits. ◇

For the real-indexed cache with the set-associative or directly-mapped replacement policy, the number of hits are not fixed under any mapping scheme (as long as the real-page assignment is unpredictable). So the cache mapping problem would be ill-defined.

Theorem: CACHE-MAPPING is NP-hard in terms of the length of the memory trace.
Proof: Suppose $T_{miss} \neq T_{noncacheable}$. The optimal cache mapping given in the proofs of Lemmas 1 through 4 remains 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 for each variable v R more times, instead of $2 * N + 1$ more times. Suppose we change the page mapping from the optimal in Lemma 1. This would increase T in the prefix by at least $R * (T_{noncacheable} - T_{hit})$, but at the same time we may decrease T in the postfix by no more than $2(S+1) * N * (T_{miss} - T_{noncacheable})$. If we 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. ◇

If we remove the mini-cache from CACHE-MAPPING, the problem becomes how to optimally select the non-cacheable virtual pages. The proofs of Lemmas 1 through 4 and the Theorem remain valid with slight adjustments. (In the proof of Lemma 1, we simply remove memory accesses to $A(v'', 1)$ from the prefix.) Hence, this special case of CACHE-MAPPING remains NP-hard.

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] 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.

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

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

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.