Technical Report UMIACS-TR-2000-60, Institute for Advanced Computer Studies
University of Maryland at College Park, September 2000.

Real-Time Memory Management: Compile-Time Techniques and Run-Time Mechanisms that Enable the Use of Caches in Real-Time Systems

Bruce L. Jacob and Shuvra S. Bhattacharyya

Department of Electrical & Computer Engineering, and
Institute for Advanced Computer Studies
University of Maryland, College Park
{blj,ssb}@eng.umd.edu

Abstract

*This paper demonstrates the intractability of achieving statically predictable performance behavior with traditional cache organizations (i.e., the **real-time cache problem**) and describes a non-traditional organization—combined hardware and software techniques—that can solve the real-time cache problem. We show that the task of placing code and data in the memory system so as to eliminate conflicts in traditional direct-mapped and set-associative caches is NP-complete. We discuss alternatives in both software and hardware that can address the problem: using address translation with software support can eliminate non-predicted conflict misses, and explicit management of the cache contents can eliminate non-predicted capacity misses. We present a theoretical analysis of the performance benefits of managing the cache contents to extend the effective size of the cache.*

Keywords: *real-time caches, memory systems for real-time systems, static & dynamic cache management*

1 Introduction

Real-time embedded systems require guaranteed performance behavior because they interact with the real world. Most of today's embedded microprocessors are yesterday's high-performance desktop processors—they were designed with instruction throughput in mind, not predictable real-time behavior. Therefore, they cannot guarantee performance behavior when using inherently probabilistic mechanisms such as caches, and so they are unsuitable for use in real-time embedded systems. As a result, real-time embedded systems often run with these mechanisms disabled; for instance, disabling the cache guarantees the performance behavior of the memory system.

It is difficult for software to make up for hardware's inadequacy. There have been numerous studies rearranging code and data to better fit into a cache: examples include cache blocking, page coloring, cache-conscious data placement, loop interchange, unroll-and-jam, etc. [McFarling 1989, Carr 1993, Bershad et al. 1994, Carr et al. 1994, Calder et al. 1998]. Additionally, it has long been known that increasing set associativity decreases cache misses. However, these mechanisms, both hardware and software, have the goal of *increasing* the number of cache hits, not *guaranteeing* the number of cache hits. Not surprisingly, guaranteeing cache performance requires considerably more work. In fact, as we show in this paper, the task of assigning memory addresses to code and data objects so as to eliminate cache conflicts is NP-complete.

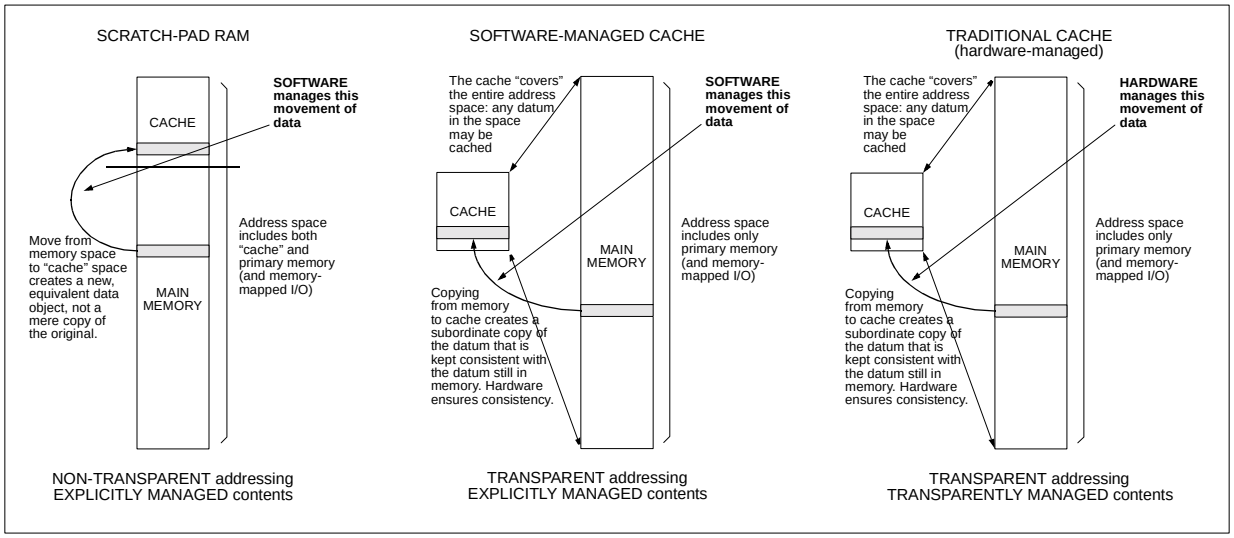

Figure 1: Examples of caches and non-caches.

This paper shows that a combination of both hardware and software—variations on traditional cache and runtime system architectures—can solve the problem. The problem reduces to two components: (1) guaranteeing that there will be no ***conflict*** misses in the cache, and (2) guaranteeing that there will be no ***capacity*** misses in the cache. ***Compulsory*** misses do not present a problem because they are statically predictable.

2 Caches vs. Fast Storage

Before we describe our modified cache architecture, we need to discuss traditional cache architectures. Many good cache primers may be found in texts on computer architecture (e.g. [Hennessy & Patterson 1996]), but the fundamental idea is that a cache brings a copy of desired data close to the processor operating on that data. It is typically built of fast storage, compared to the storage holding the original data. For instance, most processor caches are built of SRAM, and they cache copies of original data stored in DRAM.

However, we would like to introduce a strict definition of the term, because several fields use the term "cache" to mean "fast storage" and they are not necessarily the same thing. For example, the DSP world uses the word "cache" to describe the dual SRAM scratch-pads found on DSP chips that enable simultaneous access to two operands [Lapsley et al. 1994]. These differ from traditional caches in their method of data management: they do not hold a *copy* of a datum; they hold the datum itself. Caches hold *copies* of data.

The explanation is best served by illustration. Figure 1 shows three separate organizations; on the left is a DSP-style scratch-pad, on the right is a traditional cache, and in the middle is our mechanism. The scratch-pad is in a separate namespace from the primary memory system; it is *non-transparent* in that a program addresses it explicitly. A datum is brought into the scratch-pad by an explicit move that does not destroy the original copy. Therefore, two equal versions of the data remain—there is no attempt by the hardware to keep the versions *consistent* (ensure that they always have the same value) because the semantics of the mechanism suggest that the two copies are not, in fact, copies of the same datum but instead two independent data. If they are to remain consistent, it is up to software. By contrast, the traditional

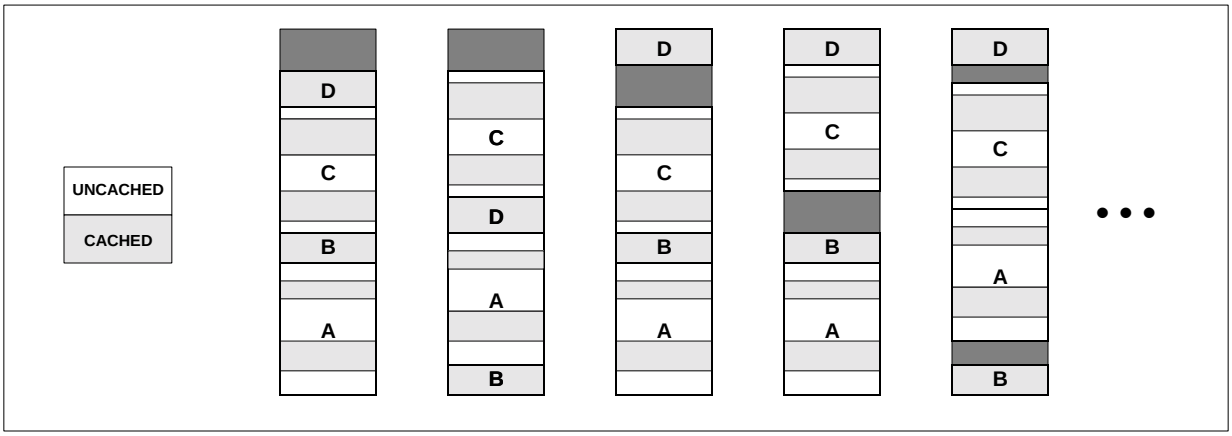

Figure 2: Possible layouts of cached and uncached objects in the memory space.

cache uses the same namespace as the primary memory system; it is *transparent* in that a program addresses main memory to access the cache—a program does not explicitly access the cache or even need to know that the cache exists. Our mechanism will be discussed later.

This, then, is our definition: A *cache* is identified by its management policy—it holds copies of data, not the data themselves; its method of addressing is transparent; and, in the case of traditional, hardware-managed caches, a program need not know it exists or invoke it explicitly to use it. Any other mechanism is not a cache. In particular, a cache is *not* identified by its implementation—i.e., just because it is built of SRAM does not make it a cache, and a cache can be built of any storage technology, not just SRAM.

3 The Cache Placement Problem and Its Complexity

Real-time embedded systems require guaranteed performance behavior because they interact with the real world. Today's embedded microprocessors are yesterday's high-performance desktop processors; they were designed with instruction throughput in mind—not predictable real-time behavior. Therefore, they cannot guarantee performance behavior, especially for inherently probabilistic mechanisms such as caches, and so they are unsuitable for use in real-time embedded systems. As a result, real-time embedded systems often run with caches disabled.

The difficult problem is eliminating conflict misses. Even if we eliminate capacity misses by judiciously selecting the items to cache so that we do not exceed the cache's storage, the task is still intractable. Assume that through code inspection or application profiling or compiler optimization it is possible to identify a subset of the program (chosen at a cache-block granularity) that should be cached. The rest of the program will remain non-cached for the duration of application execution. To guarantee that there will be no cache conflict problems during execution, it is necessary to arrange the cached items in memory such that the maximum degree of overlap in the cache is less than the cache's degree of associativity. The intuitive picture is shown in Figure 2: there are a number of atomic objects in the program labeled *A, B, C,* ... that cannot be broken up any further. Portions of these objects may be cached or uncached; this is identified by shading within each object. Assuming the total size of the regions to be cached does not exceed the size of the cache, there are a number of potential arrangements of the objects in the memory space, each of which might yield a

conflict-free cache arrangement. Finding such a conflict-free arrangement of code and data objects is a non-trivial problem. The following is a formal description:

CONFLICT-FREE CACHE PLACEMENT

INSTANCE: <memory size *M*, cache size *C*, cache associativity *A*, set *O* of memory objects, v:{block} $\rightarrow$ {0,1}>, where *M* and *C* are in units of cache blocks, $A \in Z^+ \leq C$, and an *object* is an ordered set of contiguous cache-block-sized regions. We write set *O* as follows:

$$O = (\{b_{11}, b_{12}, \ldots, b_{1n_1}\}, \{b_{21}, b_{22}, \ldots, b_{2n_2}\}, \ldots, \{b_{Z1}, b_{Z2}, \ldots, b_{Zn_Z}\}), \quad (1)$$

where b_{ij} is the *j*th block-sized portion of object *i* (termed o_i) and has the associated value $v(b_{ij})$, which is 1 or 0 depending on whether block b_{ij} is to be *cached* or *non-cached*, respectively.

QUESTION: Does there exist a realistic mapping σ:$\{o_i\} \rightarrow {Z_0}^+$ such that the degree of overlap in the cache is consistent with the cache associativity (thereby producing a memory footprint that is without cache conflicts)?

Each object o_i can be mapped via a function σ onto the memory space ${Z_0}^+ < M$. This function indicates the starting point of each object. Because the objects are collections of contiguous regions, the mapping also implicitly defines the memory location for each individual block within each object: the first block-sized region must lie at the object's memory location (i.e., $\sigma(b_{i1}) = \sigma(o_i)$).* The second block-sized region must lie at the next sequential (also block-sized) memory location (i.e., $\sigma(b_{i2}) = \sigma(b_{i1}) + 1 = \sigma(o_i) + 1$). The third block-sized region must lie at the next sequential memory location (i.e., $\sigma(b_{i3}) = \sigma(b_{i2}) + 1 = \sigma(b_{i1}) + 2 = \sigma(o_i) + 2$), etc. This places the block-sized regions of each object into *C/A* different equivalence classes $\{ E_0, E_1, \ldots E_{C/A-1} \}$ which correspond to cache sets: a region's equivalence class is the cache set to which it maps, determined by its memory location modulo the number of sets in the cache (*C/A)*. Therefore, by definition, $b_{ij} \in E_{\sigma(b_{ij}) \bmod C/A}$.†

To have a realistic mapping σ with a conflict-free cache layout, we must satisfy the following. First, the program must fit into the memory space:

$$\left(\sum_{o_i \in O} |o_i| \leq M \right) \wedge (\max(\sigma(o_i) + |o_i|) < M), \quad (2)$$

where $|o_i| = n_i$ is the cardinality of ordered set o_i and therefore size of object *i*.

Second, there must be no overlap among objects in the memory space (informally, $\forall a \neq b$, $o_a \cap o_b = \emptyset$).

$$\sigma(b_{ij}) = \sigma(b_{xy}) \Rightarrow (i = x) \wedge (j = y). \quad (3)$$

Last, for the cached objects, the number of overlaps in the cache must be consistent with the cache's degree of associativity (e.g. for a direct-mapped cache, there should be at most one cached element in each equivalence class;

---

* Here, for simplicity, we incorporate a minor abuse of notation, and extend the domain of the "memory mapping function" σ to include the set $\{ o_1 \cup o_2 \cup \ldots o_Z \}$ of individual cache-block-sized regions within the memory objects.

† Note that some cache architectures use an indexing scheme based on hash values instead of modular arithmetic; one could simply replace the **mod** in this statement with the appropriate hash function.

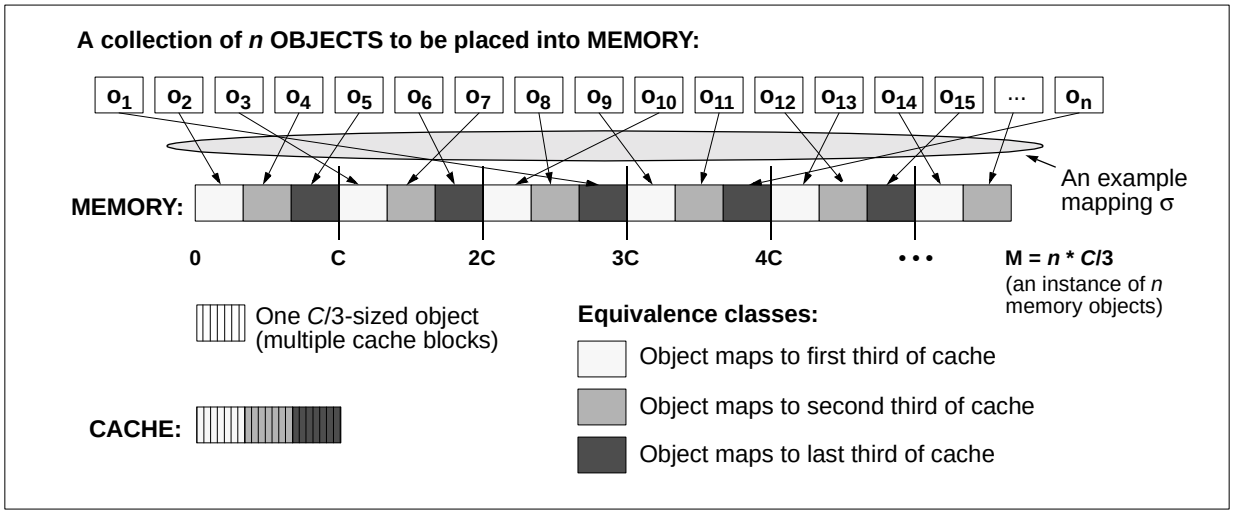

Figure 3: Special case of PLACEMENT in which all objects are equal in size, each *C*/3.

for a two-way set associative cache, there should be at most two cached elements in each equivalence class, etc.).

$$\forall k \sum_{b_{ij} \in E_k} v(b_{ij}) \leq A\,. \tag{4}$$

This decision problem (PLACEMENT for short) is NP-complete. Our first proof covers direct-mapped caches and programs whose code and data just fit into the available memory space (i.e., A=1 and $\sum |o_i| = M$). We transform an instance of 3-COLOR to an instance of PLACEMENT in which all of the objects to be placed into memory have the same size, which is 1/3 of the cache size. Note that if we restrict ourselves to those instances of the cache placement problem in which all of the data objects are size C/3 (i.e., $\forall o_i$, $|o_i| = C/3$), then there are only three places in the cache to which any object can map, since A=1 and the sum of the objects' sizes equals the memory size. This is illustrated in Figure 3. The mapping function $\sigma:\{o_i\} \rightarrow Z_0^+$ is extremely restricted by this arrangement: it can only map to a limited number of addresses in Z_0^+: 0, C/3, 2C/3, C, 4C/3, 5C/3, 2C, etc. This effectively defines three equivalence classes for objects, which correspond to the three thirds of the cache to which an object can map (note that the problem description defines C/A equivalence classes; this is slightly different). Two objects cannot be in the same equivalence class—i.e. they cannot map to the same third of the cache—if any of their blocks would overlap in the cache. This is shown in Figure 4; objects A and B have blocks in common positions that are to be cached, and thus they cannot map to the same third of the cache. This restricts where in memory they may be placed relative to each other. Object C also has cached blocks, but they do not align with cached blocks in A or B. Object C can therefore be placed anywhere in memory relative to A and B.

An instance of 3-COLOR is an undirected graph $G = \{V, E\}$. The goal is to find a mapping $c:V \rightarrow \{0,1,2\}$ such that no adjacent vertices are assigned the same value under the mapping ($c(u) = c(v) \Rightarrow (u,v) \notin E$). Such a mapping is called a *coloring* of G. This problem can also be described in more general terms as defining three equivalence classes and placing each vertex into an equivalence class following rules that identify which vertices can be in the same equivalence class at the same time (the set of rules is the set E of edges). This wording mirrors that of PLACEMENT; we have a direct correspondence between the two problems.

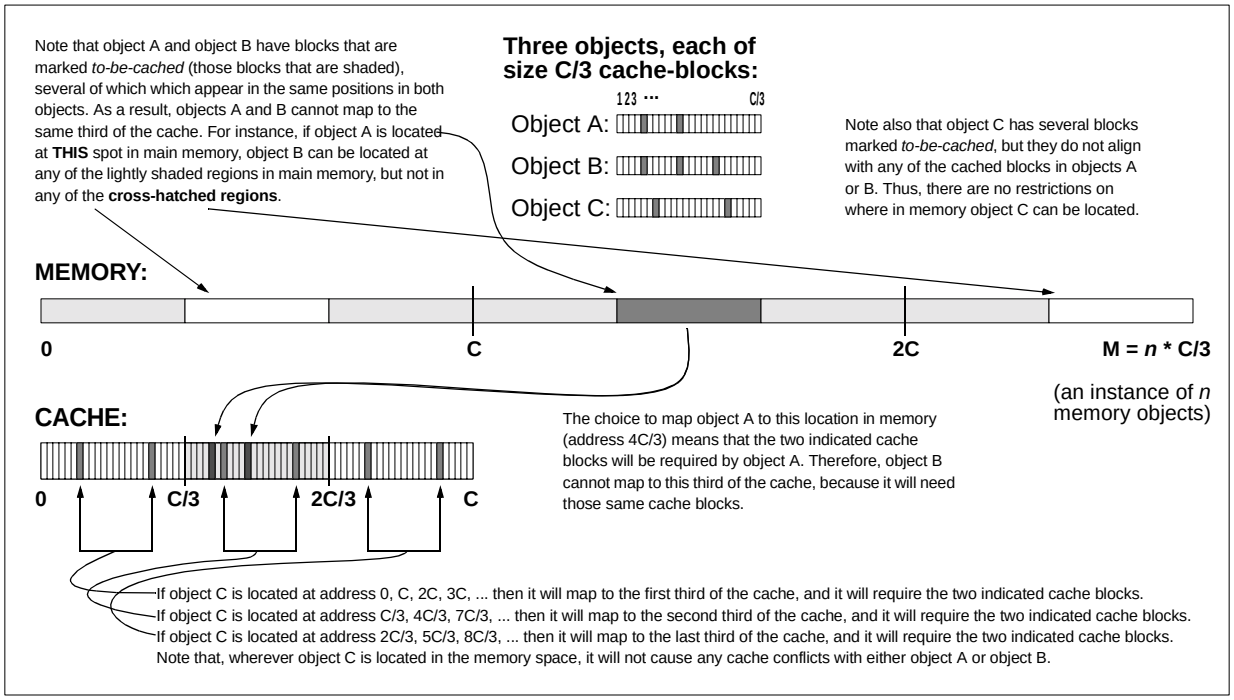

Figure 4: Cache conflicts, and conflict avoidance by judicious placement in memory.

Given an instance of 3-COLOR (an undirected graph $G_0 = \{V_0, E_0\}$), we construct an instance of PLACEMENT as follows. First, we add $2 \times |V_0|$ disconnected vertices to graph G_0 and form graph $G = \{V, E\}$, with $|V| = 3 \times |V_0|$. This does not make solving the 3-COLOR problem for graph G any easier than solving for graph G_0, because any solution for G produces a solution for G_0 by throwing away the extra vertices. The reason for adding these vertices will be clear at the end of the proof. Second, we define values for M, C, and A: the memory size M is $|V|^3$; the cache size C is $3 \times |V|^2$; the cache associativity A is 1. Last, we create set O of $|V|$ memory objects, each of size $|V|^2$. As described above, O would normally be a collection of ordered sets labeled as follows:

$$O = \{ \{b_{11}, b_{12}, \ldots b_{1\,|V|^2}\}, \{b_{21}, b_{22}, \ldots b_{2\,|V|^2}\}, \ldots \{b_{|V|\,1}, b_{|V|\,2}, \ldots b_{|V|\,|V|^2}\} \}.$$

However, to make the transformation from 3-COLOR more intuitive, we rename the $|V|^2$ blocks in each memory object, using three subscripts per block instead of two, so that each cache block within each memory object uniquely represents one of the potential $|V|^2$ edges in graph G^*:

$$\begin{aligned} o_i = \{ \; & b_{i\,1\,1}, b_{i\,1\,2}, \ldots b_{i\,1\,|V|}, \\ & b_{i\,2\,1}, b_{i\,2\,2}, \ldots b_{i\,2\,|V|}, \\ & \ldots, \\ & b_{i\,i\,1}, b_{i\,i\,2}, \ldots b_{i\,i\,|V|}, \\ & \ldots, \\ & b_{i\,|V|\,1}, b_{i\,|V|\,2}, \ldots b_{i\,|V|\,|V|} \}. \end{aligned}$$

* This labeling ignores the fact that there are actually only $(n^2-n)/2$ potential edges in a nondirected graph: a connectivity matrix is usually an upper triangular matrix without the diagonal elements. The labeling system is chosen more for ease of representation than for efficiency.

The naming convention is chosen intentionally to look like the connectivity matrix for the set E of edges; we will show why in a moment. Written in a linear form, the entire collection O of objects looks like the following:

$O = \{ o_1, o_2, ..., o_i, ..., o_{|V|} \} =$

$\{b_{111}, b_{112}, ... b_{11\,|V|}, b_{121}, b_{122}, ... b_{12\,|V|}, \cdots, b_{1\,i1}, b_{1\,i\,2}, ... b_{1\,i\,|V|}, \cdots, b_{1\,|V|\,1}, b_{1\,|V|\,2}, ... b_{1\,|V|\,|V|}\},$

$\{b_{211}, b_{212}, ... b_{21\,|V|}, b_{221}, b_{222}, ... b_{22\,|V|}, \cdots, b_{2\,i\,1}, b_{2\,i\,2}, ... b_{2\,i\,|V|}, \cdots, b_{2\,|V|\,1}, b_{2\,|V|\,2}, ... b_{2\,|V|\,|V|}\},$

...,

$\{b_{i\,11}, b_{i\,12}, ... b_{i\,1\,|V|}, b_{i\,21}, b_{i\,22}, ... b_{i\,2\,|V|}, \cdots, b_{i\,i\,1}, b_{i\,i\,2}, ... b_{i\,i\,|V|}, \cdots, b_{i\,|V|\,1}, b_{i\,|V|\,2}, ... b_{i\,|V|\,|V|}\},$

...,

$\{b_{|V|\,11}, b_{|V|\,12}, ... b_{|V|\,1\,|V|}, b_{|V|\,21}, b_{|V|\,22}, ... b_{|V|\,2\,|V|}, \cdots, b_{|V|\,i1}, b_{|V|\,i2}, ... b_{|V|\,i\,|V|}, \cdots, b_{|V|\,|V|\,1}, b_{|V|\,|V|\,2}, ... b_{|V|\,|V|\,|V|}\}$

This naming convention allows us to intuitively assign to each block in each memory object an appropriate value $v(b)$ (1/0 for CACHED/NON-CACHED), derived from the connectivity matrix. If there is an edge between vertices i and j, then the value '1' is found in the connectivity matrix at positions (i,j) and (j,i). The same is true for the blocks in a memory object—if there is an edge between vertices i and j, then vertices i and j cannot be in the same equivalence class—their corresponding memory objects cannot map to the same third of the cache; we ensure this by marking block (i,j) to be cached in memory objects i and j. We thus set the values $v(b)$ according to the edges in E: unless otherwise specified, a block-sized region of a memory object is non-cached, i.e. $v(b) = 0$. However, for each edge $(i, j) \in E$, we have the following:

Blocks $b_{i\,i\,j}$ and $b_{j\,i\,j}$ are both cached, i.e. $v(b_{i\,i\,j}) = v(b_{j\,i\,j}) = 1$.

Because G is an undirected graph, and because we are looking at a full matrix and not an upper triangular matrix, there is some symmetry: if $(i, j) \in E$, then by definition $(j, i) \in E$, and we effectively have the following for an edge between vertices v_i and v_j :

Blocks $b_{i\,i\,j}$, $b_{i\,j\,i}$, $b_{j\,i\,j}$, and $b_{j\,j\,i}$ are all cached, i.e. $v(b_{i\,i\,j}) = v(b_{i\,j\,i}) = v(b_{j\,i\,j}) = v(b_{j\,j\,i}) = 1$.

This arrangement is illustrated in Figure 5. The figure shows three objects o_i, o_j, o_k, corresponding to vertices v_i, v_j, v_k. The graph contains edges (i, j) and (i, k) but not edge (j, k). Therefore vertices v_i and v_j cannot have the same color, and vertices v_i and v_k cannot have the same color. This arrangement creates three objects o_i, o_j, o_k, such that o_i and o_j cannot align to the same third of the cache, and neither can o_i and o_k. Any legal coloring of the vertices corresponds to a legal mapping of the objects to the memory space, and any legal mapping of the objects in the memory space corresponds to a legal coloring of the vertices in the graph.

To obtain the corresponding solution to the 3-COLOR problem instance, given a solution to the PLACEMENT instance, we eliminate the extra vertices added to graph G_0 and obtain values for the color mapping function as follows:

$$c(v_i) = \frac{\sigma(o_i) \bmod C}{C/3} \tag{5}$$

The color value for vertex v_i corresponds to whether memory object o_i maps to the first, second, or last third of the cache. All vertices whose corresponding memory object maps to the first third of the cache have color = 0. All vertices whose corresponding memory object maps to the second third of the cache have color = 1. The rest of the vertices have

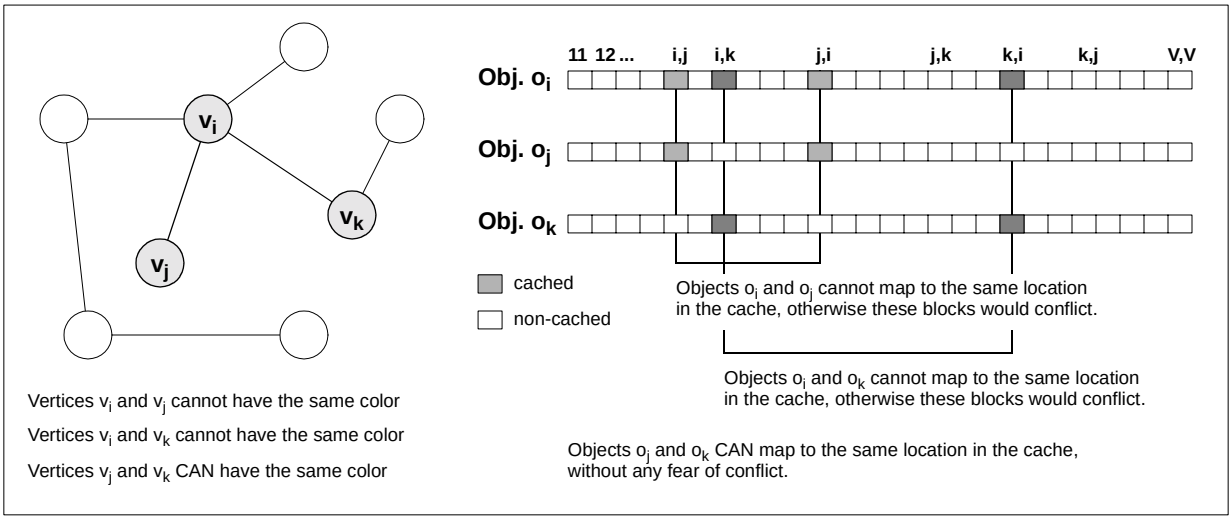

Figure 5: Correspondence between graph vertices and memory objects.

color = 2. Because σ is so constrained in its choice of mappings (by construction of the PLACEMENT instance), the right-hand side of Eq. (5) is guaranteed to produce integer values {0,1,2}.

Note that by definition, the PLACEMENT algorithm will evenly distribute all memory objects across the three equivalence classes. Thus, a given graph may have so many connected vertices that, even though it is possible to assign a particular vertex to a particular color, there might not be room to place the corresponding memory object in a memory location that maps to the appropriate third of the cache. This is why we added $2 \times |V|$ disconnected vertices to graph G_0 before transforming the problem to an instance of PLACEMENT. Doing so triples the size of the memory space, ensuring that there is enough room in memory to assign all memory objects corresponding to vertices in graph G_0 to the same third of the cache, if possible—this would correspond to a graph G_0 in which it is possible to give all vertices the same color. Had we transformed the problem from G_0 directly, the corresponding objects would map to contiguous locations in the memory space, ensuring that the objects are spread evenly across the available equivalence classes—ensuring that it would not be possible to place all vertices in G_0 in the same equivalence class. By adding the extra disconnected vertices (which by definition correspond to memory objects that have no cached blocks), we ensure that there is more than enough room in the memory space to assign any memory object in G_0 to any possible equivalence class.

Note that a valid solution to the PLACEMENT instance will be found iff there is a valid solution to the 3-COLOR instance:

1. Eq. (2) is satisfied by construction: the objects fit exactly into the memory space.
2. Eq. (3) is satisfied by the PLACEMENT algorithm, which will not generate an invalid mapping.
3. Eq. (4) may or may not be satisfied, depending on the problem instance. Remember that we have restricted ourselves to a subset of the PLACEMENT problem space in which $A = 1$. Therefore, a valid solution can have only one block in each of the C equivalence classes $\{ E_0, E_1, \ldots E_{C-1} \}$. By construction, two memory objects may map to the same third of the cache iff they do not have blocks in the same position that are to be cached, i.e. iff their corresponding vertices have no connecting edge. This means that the three sets of memory objects,

each of which maps to a different third of the cache, correspond to three sets of vertices in G in which no two vertices are incident to a common edge. If the PLACEMENT algorithm can map two objects to the same third of the cache, then the corresponding two vertices must not have a common edge. If the PLACEMENT algorithm cannot map two objects to the same third of the cache, then the corresponding two vertices must have a common edge. If two vertices have a common edge, their corresponding memory objects cannot map to the same third of the cache. If two vertices have no common edge, their corresponding memory objects may map to the same third of the cache. Therefore Eq. (4) is satisfiable iff G is 3-colorable.

To finish the proof, we note that the transformation from 3-COLOR to PLACEMENT can be performed in polynomial time, as can the transformation back from a PLACEMENT solution to a 3-COLOR solution. Finally, note that a solution for PLACEMENT can be verified in polynomial time. Eqs. (2) and (4) can all be checked in time O(n), where n is the total number of blocks in the program. Eq. (3) can be checked in time proportional to max(M, n). ***Q.E.D.***

Though it is an integral part of the proof, the fact that the memory size equals the sum of the individual object sizes is not what makes the problem intractable. Increasing the size of the memory space does not make the problem solvable in polynomial time. To prove this, we map an instance of k-COLOR to PLACEMENT in which there are (k-1) extra blocks in the memory space, above and beyond the amount needed to hold the memory objects.

Given an instance of k-COLOR, an undirected graph $G = \{V, E\}$ together with a positive integer $k \geq 3$, we construct an instance of PLACEMENT as follows. First, we define M, C, and A: the memory size M is $(2k\text{-}1) \times |V|^3 + (k\text{-}1)$; the cache size C is $(2k\text{-}1) \times |V|^2$; the cache associativity A is 1. Then we create set O of $|V|$ memory objects, each of size C = $(2k\text{-}1) \times |V|^2$. As in the proof above, the objects in O would be labeled so as to indicate individual (potential) edges in E. That is, each o_i contains the blocks

$$
\begin{aligned}
o_i = \{\ & b_{i\,1\,1}, b_{i\,1\,2}, \ldots b_{i\,1\,|V|}, \\
& b_{i\,2\,1}, b_{i\,22}, \ldots b_{i\,2\,|V|}, \\
& \ldots, \\
& b_{i\,i\,1}, b_{i\,i\,2}, \ldots b_{i\,i\,|V|}, \\
& \ldots, \\
& b_{i\,|V|\,1}, b_{i\,|V|\,2}, \ldots b_{i\,|V|\,|V|}\ \}.
\end{aligned}
$$

However, this accounts for only $|V|^2$ blocks per memory object. The other $2(k\text{-}1) \times |V|^2$ blocks sit between each of these blocks and act as buffers. For example, for $k = 3$, we have the following:

$$
\begin{aligned}
o_i = \{\ & X, X, b_{i\,1\,1}, X, X, X, X, b_{i\,1\,2}, X, X, \ldots X, X, b_{i\,1\,|V|}, X, X, \\
& X, X, b_{i\,2\,1}, X, X, X, X, b_{i\,22}, X, X, \ldots X, X, b_{i\,2\,|V|}, X, X, \\
& \ldots, \\
& X, X, b_{i\,i\,1}, X, X, X, X, b_{i\,i\,2}, X, X, \ldots X, X, b_{i\,i\,|V|}, X, X, \\
& \ldots, \\
& X, X, b_{i\,|V|\,1}, X, X, X, X, b_{i\,|V|\,2}, X, X, \ldots X, X, b_{i\,|V|\,|V|}, X, X\ \}
\end{aligned}
$$

Each of the blocks marked X is a buffer block and will never hold cached data (its value $v(X)$ will be zero). Therefore, we do not assign individual labels to each of these blocks. We assign values $v(b_{xyz})$ for each of the labeled blocks as in

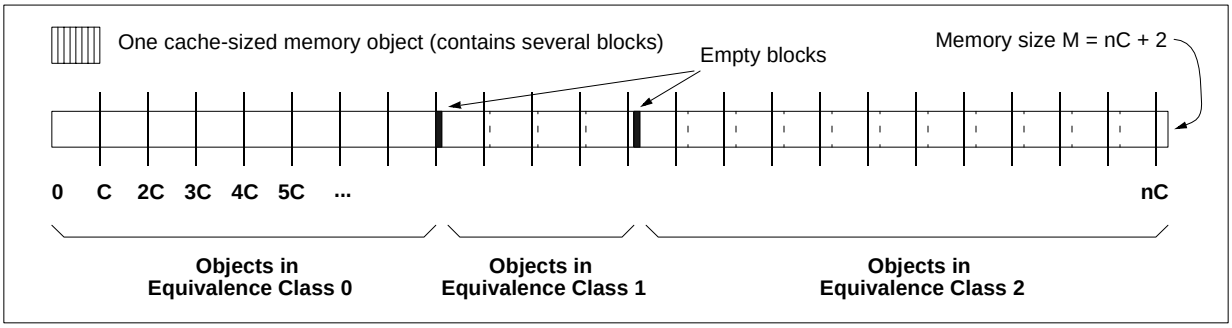

Figure 6: An example memory arrangement corresponding to *k*-COLOR in which *k* = 3, *n* = |V|.

the last proof: if $(i, j) \in E$, then $(j, i) \in E$, and we have the following for an edge between vertices v_i and v_j :

Blocks $b_{i\,i\,j}$, $b_{i\,j\,i}$, $b_{j\,i\,j}$, and $b_{j\,j\,i}$ are all cached, i.e. $v(b_{i\,i\,j}) = v(b_{i\,j\,i}) = v(b_{j\,i\,j}) = v(b_{j\,j\,i}) = 1$.

At this point, the instance of PLACEMENT is complete. The question that remains is how its solution efficiently induces a solution for the corresponding graph coloring instance. In the last proof, the three equivalence classes that corresponded to graph colors were the three thirds of the cache to which an object could map. In this proof, the objects are all the same size as the cache. In the last proof, we added extra non-cached memory objects to the instance (the extra disconnected vertices in the graph) in order to allow free association of memory objects with the three equivalence classes. In this instance construction, the freedom to move objects around comes from the extra non-labeled/non-cached blocks that are added to each object, and the equivalence classes to which the objects are assigned correspond to whether the object is located at memory location 0 (mod C), 1 (mod C), ..., or (*k*-1) (mod C).

These equivalence classes illustrated in Figure 6, for $k = 3$. There are as many memory objects in the memory space as there are vertices in the graph ($n = |V|$). Each memory object is the size of the cache. The memory size M is equal to the sum of the object sizes plus (k - 1) extra cache blocks. In this example, $M = nC + 2$. The arrangement is such that there are three regions of memory objects laid out in the memory space, defined by where the two blank spaces are positioned (the empty blocks are assumed to be aligned on cache-block-sized boundaries). After each empty block, the subsequent memory objects are all offset from the previous memory objects by one cache block (mod C). There are n memory objects in the memory space; those in the lower address space are aligned on cache-sized boundaries, and the dotted lines in the figure indicate the locations of memory objects that are found at higher addresses in the memory space.

How does the assignment of the $v(b)$ values help solve the *k*-COLOR problem? As before, presence of an edge between two vertices indicates that the two vertices may not belong to the same equivalence class. By making the corresponding blocks in each memory object cached, we ensure that those memory objects cannot lie in the same equivalence class at the same time. However, we have to make sure that if two memory objects are offset by 1 or more blocks in the memory space (corresponding to different equivalence classes), doing so does not inadvertently make their blocks marked *to-be-cached* conflict with other blocks marked *to-be-cached* that have nothing to do with the edge in question. This is where the extra "buffer" blocks in each memory object come in useful. They ensure that any two memory objects whose corresponding graph vertices have an edge in E cannot map to the same equivalence class, while the same two memory objects can map to any different equivalence classes without fear that those blocks marked *to-be-*

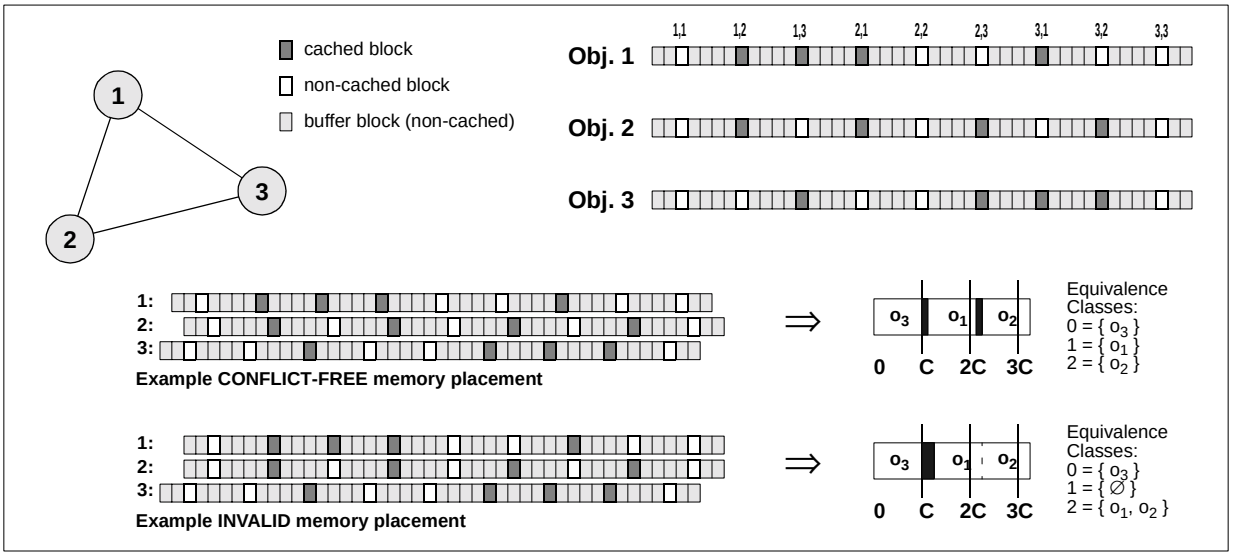

Figure 7: Simple example of *k*-COLOR transformation for *k* = 3, |*V*| = 3.

cached will interfere with blocks in other memory objects.

Figure 7 provides an example, again with $k = 3$. Here, the number of vertices is very small (3) to simplify the illustration. Because the graph is fully connected, every edge has two corresponding blocks in each memory object that are marked *to-be-cached*. Thus, none of the memory objects can align in the cache: each has to be offset by at least one block from every other memory object. Since there are $(k - 1)$ buffer blocks on each side of every labeled block in the memory objects, it is possible to offset memory objects a maximum of two from each other—by construction, that is the maximum offset possible: the memory system contains $nC + 2$ blocks and must hold n objects, therefore there are only two available empty blocks in the memory system, and the maximum offset (modulo C) is two. The figure gives two example placement results: one invalid (the algorithm would not return this as a result) and one valid.

To obtain the graph coloring, given a valid mapping σ, we use the following:

$$c(v_i) = \sigma(o_i) \bmod C\,. \tag{6}$$

The color corresponds to the degree of offset a memory object is given relative to the other objects in the set.

To finish the proof, we note that the transformation can be accomplished in polynomial time, and the validity of a PLACEMENT solution can be checked in polynomial time. ***Q.E.D.***

4 Conflict Misses: Virtual Addressing = Full Associativity

We have shown two important things:

1. To obtain real-time performance from caches, one must eliminate conflict misses.
2. Eliminating conflict misses in traditional, hardware-managed caches is intractable.

The problem is solvable, however. Whereas the problem is difficult for either hardware or software mechanisms in isolation, a combination of both hardware and software—variations on traditional cache and runtime system architectures—can provide predictable real-time behavior from caches. As mentioned previously, though fully

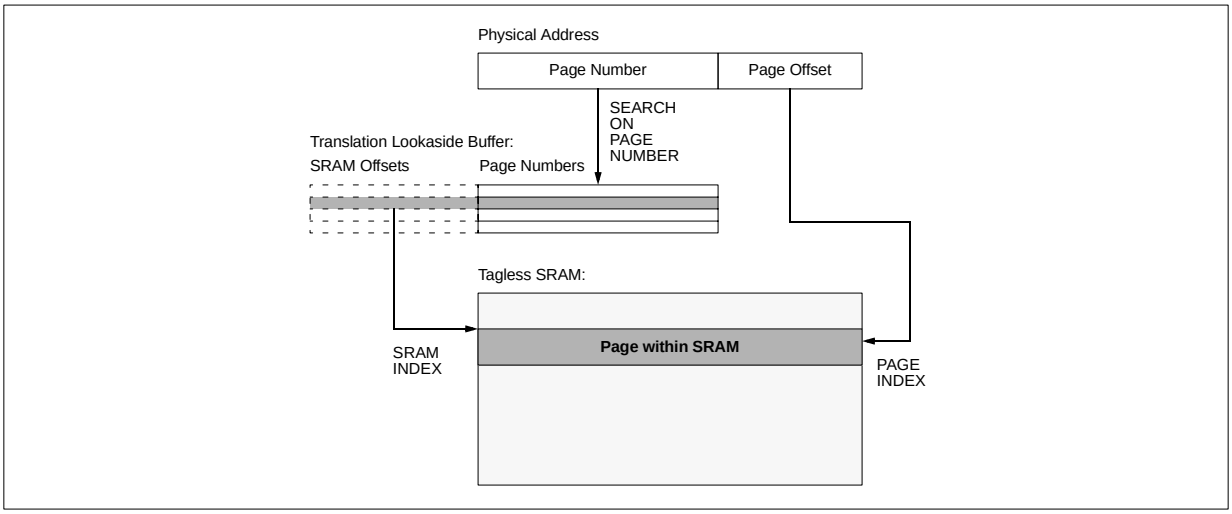

Figure 8: A real-time cache architecture.

associative caches would solve conflict misses, traditional designs are too expensive to implement in low-power embedded systems, as fully associative caches burn too much power to be useful at large sizes (several kilobytes or larger). The alternative is a *real-time cache*: a software-managed fully associative cache with extremely large cache blocks (the use of larger block sizes reduces the number of tags and thus power). To address capacity issues (i.e. to increase the effective size of the cache), one can dynamically and predictably manage the cache contents.

The real-time cache is a design that disassociates naming from storage in the same vein as virtual memory [Jacob & Mudge 1998a, Jacob & Mudge 1998b, Jacob & Mudge 1998c]. This allows one to freely locate objects in a tagless SRAM without having to change their location in physical memory. Objects are relocated at the granularity of pages, which are on the order of 100 bytes (128 bytes, 256 bytes, etc.). To provide real-time guarantees, every page of the SRAM is mapped in hardware. The architecture is illustrated in Figure 8. A translation lookaside buffer (TLB) maps the contents of a tagless SRAM: if a chunk of data is in the SRAM, its mapping is held in the TLB. As in normal TLBs, the search is fully-associative—the TLB is a content-addressable memory (set-associative TLBs are possible and simply limit one's flexibility in placing items in the memory space but are still more flexible than an ordinary cache). The difference is that the equivalent of the page frame number (the "SRAM offset") is not actually held in the TLB. It is inferred from the slot number. Therefore, if the matching page number is found in slot 0, it refers to the first page within the SRAM, etc. The bottommost bits of the physical address are an offset into this page. If the page number is not found in the TLB, it is assumed that it is a valid physical address, and the address is sent to the primary memory system as-is. Thus, every physical address that an application generates may or may not reference an item held in the SRAM, and the caching is transparent.

Clearly, this organization supports the static management of data. One need only choose the subset of code and data that is to be held in the SRAM, copy that code and/or data into the SRAM, and initialize the TLB appropriately. From that moment on, all references to the cached information would go to the SRAM, not main memory. For an SRAM of 8Kbytes and a granularity of 256 bytes (the page size), the TLB would have 32 entries. Larger page sizes can be used to reduce the size of the TLB, at the expense of flexibility. This organization is also much more flexible than a scratch-pad RAM (i.e. the cache organization typically found in DSP architectures, in which an item's address determines the

memory structure in which it is held [Lapsley et al. 1994]) in that it allows the arbitrary arrangement of data at a page-sized granularity, without regard to contiguity or inter-object distances. For example, items that are adjacent in the memory space can be cached or not cached without creating any cache conflicts or addressing problems.

Note that this architecture is logically equivalent to a fully associative cache (a CAM) with an unusually large block size. The differences are that the system uses physical addresses, not virtual addresses, and the cache tags in this organization are loaded by software, not by hardware (much like a software-managed TLB).

Note also that there is another way to make the cache placement problem solvable in polynomial time, assuming that the cached/non-cached regions have a simple organization (for example, one cached region per atomic object): if M is large enough, then we can simply choose mappings from $\{s_{i1}\}$ to Z^+ such that each cached region begins exactly one cache block beyond the previous object's cached region. This fails to work when we have odd organizations, such as two arrays, one in which every other block should be cached, the other in which every third block should be cached. However, if the application has well-behaved cached/non-cached regions, M can be increased to (almost) arbitrary size by using virtual addressing. Hardware support includes a software-managed cache [Jacob & Mudge 1998a] or a traditional TLB+cache, provided that the TLB is software-managed and fully maps the cache with at least one slot to spare, using the uppermost TLB slots for cached data and the remaining TLB slot/s for translating the rest of the application in memory. This scheme requires translation for *all* memory locations, not just *cached* memory locations, which means we need a small memory space, a large TLB, a large page size, or a well-thought-out translation scheme.

5 Capacity Misses: Real-Time Cache Management

If the cache is statically managed—that is, the decision of what to cache or not cache is made statically, and the contents of the cache do not change during program execution—then all addresses are either cached or non-cached, and both types of access have absolutely deterministic access times. The challenge is to determine at compile time (or just-in-time before program execution) what will be the most important page-sized regions of memory, and to rearrange code and data so that a page's contents hold either "hot" data or "cold" data, but not a combination of both, otherwise the cache would contain cold data that is referenced infrequently. These are jobs for a compiler and code profiler, along the lines of work done on DSP dataflow models and memory buffer usage [Lee & Messerschmitt 1987, Bhattacharyya et al. 1996].

However, what if we cannot fit everything we want cached into the cache? The previous section solves conflict issues for us, but what about capacity issues? The cache can also be dynamically managed—Figure 9 illustrates an idea similar to virtual memory. All code and data in a real-time application (as well as the RTOS itself) is categorized:

1. Always to be cached
2. Never to be cached
3. Exhibits periodic locality

The first two categories speak for themselves. Hopefully, the size of category #1 is smaller than the SRAM itself. If so, the third category represents items that exhibit predictable burstiness in their locality—items that are used repeatedly for a short duration and then not referenced at all for a long period. Examples that immediately come to mind include loop

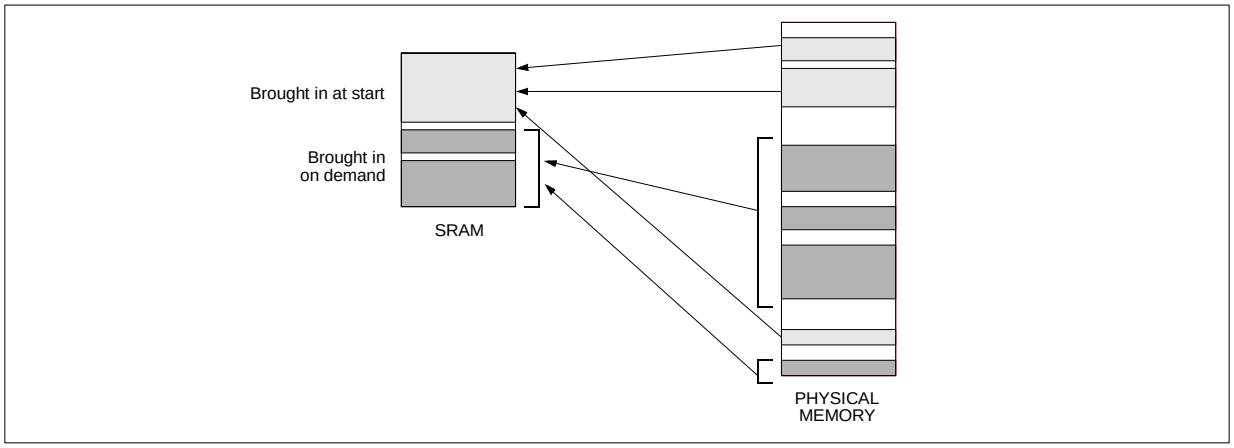

Figure 9: Dynamic management of the real-time cache architecture.

code and data. These items should be cached, but only while the loop is active—at other times the items should not occupy cache space.

The real-time management of these items is effected by placing code at their beginning and endpoints. We will call these code blocks the start-block and the end-block. The start-block brings the loop code and data (at least, that of the loop code and data that will be cached) into the cache, and sets the TLB appropriately. The end-block unmaps the cached items from the TLB and writes out any data to the memory space that requires it.

Since the size of loops is usually known in advance for many DSP subsystems, the execution time for the entire block (including start-block and end-block regions) can often be calculated statically. In such cases, this cache management routine is deterministic and offers real-time memory management. The issues to explore are how to perform system-level partitioning of code and data into always-cached, never-cached, and periodically-cached components; and how to dynamically reconfigure such partitionings as an application's behavior or environment changes (e.g., changes in the set of active subsystems, overall processing load, quality of service requirements, and power consumption constraints).

6 Theoretical Performance Analysis

An LRU-Managed CAM Is Better Than a Statically-Managed CAM Is Better Than a Scratch-Pad RAM

Using an analytical approach to cache models similar to that presented in previous work [Jacob et al. 1996], we show that, for many applications, a perfect statically managed CAM performs worse than a simple LRU-managed CAM. The static-CAM results represent an upper bound on realistic scratch-pad RAM performance. A dynamically-managed CAM can be used to solve the problem of **capacity** misses, and we briefly present a software architecture for real-time cache management. This analysis compares the static management of a scratch-pad RAM (a hardware mechanism used in nearly all DSPs [Lapsley et al. 1994]) to the dynamic management of a fully associative cache. The scratch-pad is modeled as a CAM, yielding an upper bound on performance. The analysis shows that somewhere between no locality and perfect locality there is a crossover where it becomes more efficient to manage the cache dynamically.

Assume that we have a running application that generates a stream of references to locations in its address space. Let us say that a function *f(x')* yields the number of times the application referenced address *x'*. Note that we have not

specified the granularity of access: it could be at a byte, word, or block granularity; the only restriction for this analysis is that all references have the same granularity.

Suppose that we sort the pairs { *x'*, *f(x')* } from highest *f(x')* value to lowest *f(x')* value, thereby creating a new relation. If we map the reordered *x'* domain onto the integers via mapping $x : Z^+ \rightarrow x'$, we can imagine the new function *f(x)* yielding the frequency of access of the x^{th} most commonly referenced item. That is, *f(1)* is the number of times the most commonly referenced item was referenced; *f(2)* is the number of times the second most commonly referenced item was referenced; *f(3)* is the number of times the third most commonly referenced item was referenced; etc. Then, assuming the following:

- perfect static management, and
- no constraints on contiguity or cache blocking,

we can organize our code and data such that the most often referenced code and data are assigned to the scratch-pad, up to the point that the scratch-pad is filled, and all other code and data reside in DRAM. Then the number of hits to a scratch-pad RAM of size *S* is given by:

$$\int_0^S f(x)dx \tag{7}$$

The total access time is then given by the following, which ignores the cost of compulsory misses (the cost of first copying everything from DRAM or ROM into the scratch-pad RAM). Note that t_S and t_D represent the access times for the scratch-pad RAM and DRAM, respectively.

$$T_{STAT} = t_S \int_0^S f(x)dx + t_D \int_S^\infty f(x)dx \tag{8}$$

This is the best performance that one can possibly achieve with a statically-managed scratch-pad RAM. This performance number is very aggressive, because it assumes that one can juggle code and data around without fear of addressing conflicts in the scratch-pad RAM, and without having to preserve contiguity. As described previously, this is not the case: it is difficult and often impossible to cache only portions of a data structure, or to place code and data at arbitrary locations in the cache. Equation (8) therefore represents the upper bound on performance for a statically managed scratch-pad RAM.

We compare this to the cost of managing a similar-sized memory in a least-recently-used (LRU) fashion [Smith 1982]. From the original application's dynamic execution, we can also derive a function *g(x)* which represents for each address *x* the average stack depth at which *x* is found. This is similar to, but not equal to, the stack-depth function used in [Jacob et al. 1996]. Like *f(x)*, the function *g(x)* is sorted, but it is sorted from least to highest value. Then the number of hits to a cache managed in an LRU fashion is given by the following:

$$\int_0^{g^{-1}(S)} f(x)dx \tag{9}$$

The limits of integration cover all addresses that have an average stack depth of *S* or less. By definition, references to

these addresses will hit in the cache, on average. From this, we can calculate the total access time, as before:

$$T_{DYN} = t_S \int_0^{g^{-1}(S)} f(x)dx + t_D \int_{g^{-1}(S)}^{\infty} f(x)dx \tag{10}$$

The dynamic scheme is better than the perfect static scheme whenever $T_{DYN} < T_{STAT}$, i.e. whenever we have the following:

$$\left[t_S \int_0^{g^{-1}(S)} f(x)dx + t_D \int_{g^{-1}(S)}^{\infty} f(x)dx \right] < \left[t_S \int_0^{S} f(x)dx + t_D \int_{S}^{\infty} f(x)dx \right] \tag{11}$$

Because $t_S \ll t_D$, the dynamic scheme is better whenever $g^{-1}(S) > S$, or, assuming g is invertible, whenever $g(S) < S$. The next step is to find $g(x)$.

In the best possible scenario, there is total locality. This is equivalent to focusing on one location at a time. An example address stream would be the following:

AAAAAABBBBBBBCCCCCCDDDDEEFFGGGGHHHHHHHHHH ...

Here, if we ignore compulsory misses (as in the previous analysis), every access to every location will be found at a stack depth of zero (there are zero intervening unique addresses). Thus, for this example, we find the following:

$$g(x) = 0 \tag{12}$$

Clearly, for any realistic S, $g(S) < S$. Therefore, for an address stream displaying this degree of locality, one should always manage the cache dynamically in an LRU fashion, not statically.

In the worst-case scenario, there is as little locality as possible. This happens when references to each address are spread out as distantly as possible, so that there is as much intervening data as possible. If we take every reference and spread it evenly across the address stream, we get the greatest average stack-depth, and we can define $g(x)$ as follows. First, we can define the average distance between successive references to the same item by simply dividing the total number of references in the dynamic execution by the number of references to address x:

$$\frac{\int_0^{\infty} f(y)dy}{f(x)} \tag{13}$$

This does not give us the stack distance, however, because the stack distance is the number of *unique items* between successive references to the same item. Equation (13) gives us the number of *total items* between successive references. Given a particular address x, for every $z < x$, there will be exactly one unique occurrence of z between successive references to x. For every $z > x$, the average number of occurrences between successive references to x will be given by

$$\frac{f(z)}{f(x)} \tag{14}$$

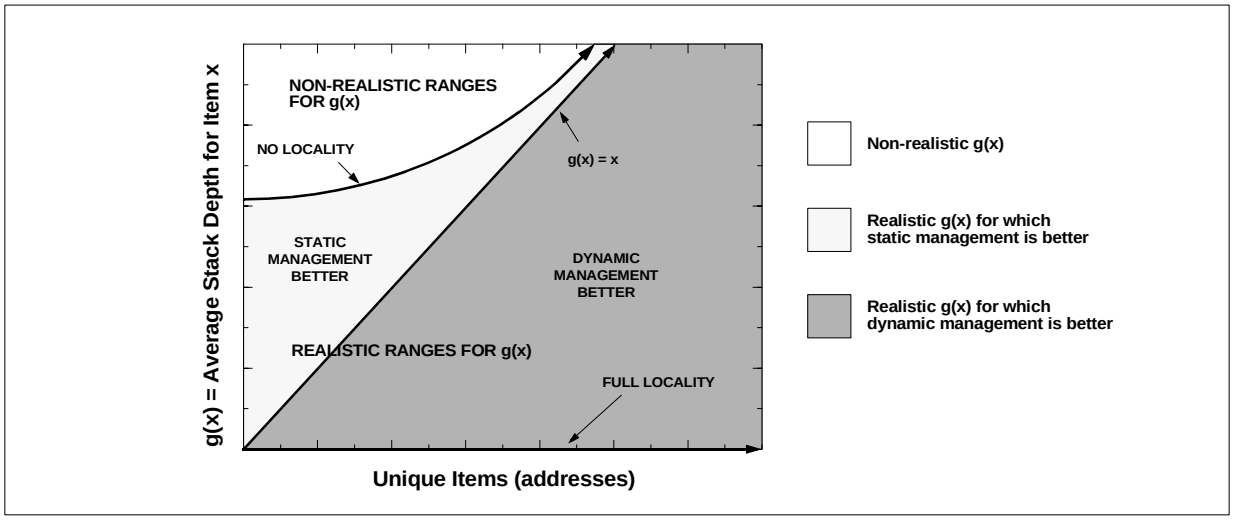

Figure 10: The use of software-managed caches in a real-time system.

What we want is the following summation of a discrete function:

$$g(x) = \sum_{z=0}^{\infty} \begin{cases} z \le x & 1 \\ z > x & \frac{f(z)}{f(x)} \end{cases} \tag{15}$$

This translates well to our continuous-function framework, yielding the following for *g(x):*

$$g(x) = x + \frac{\int_{x}^{\infty} f(z)\,dz}{f(x)} \tag{16}$$

Clearly, for all realistic *S*, *g(S)* > *S*, and we obtain the predictable result that when there is no locality, it is best to manage the scratch-pad RAM statically (assuming that our static management policy is perfect).

As a result of this analysis, we have two bounds: to a first-order approximation, *g(x)* can never be better than Equation (12), and it can never be worse than Equation (16). We have measured numerous benchmarks, and have found that *f(x)* is typically a decreasing function with a polynomial shape. This gives us a wide range of possibilities that are illustrated in Figure 10. The graph shows two things: first, there are quite a few realistic ranges for *g(x)* that exhibit better performance using dynamic management than with static management. Second, and more importantly, Equation (16) gives a form for *g(x)* that is a decreasing polynomial added to the line *y=x*. This is illustrated quite well in the figure. There is decreasing difference between static management and dynamic management as *x* increases, which means that as scratch-pad RAMs increase in size, the difference between worst-case dynamic management and best-case static management narrows.

It must be pointed out that best-case static management, as described, is impossible to achieve with any existing architecture. The analysis assumes that any chunk of data or instruction code can be located anywhere in the scratch-pad RAM, within the constraints of some given granularity. This means that a sequential array might be non-contiguous in the scratch-pad RAM. It means that the code for a function might be non-continuous in the scratch-pad RAM. A portion

of the array or a portion of the function's code might be non-cached. Therefore, one cannot sweep through the array by incrementing pointers, and similarly, incrementing the program counter may or may not arrive at the logically following instruction.

7 Conclusions

This paper has presented a preliminary exploration of cache management for real-time systems. We have defined the cache placement problem, which is the problem of placing code and data in the memory system so as to eliminate cache conflicts entirely. We have shown that this problem is intractable even in some very restricted contexts, and we have discussed a combined hardware/software approach that addresses this high complexity. This architecture for real-time cache management provides improved flexibility over existing memory management techniques in real-time systems (e.g., scratch pad RAMs in DSP processors). Imminent directions for further work involve experimental exploration of the concepts introduced in this paper, and the development of compiler technology that exploits the proposed architecture.

References

B. N. Bershad, D. Lee, T. H. Romer, and J. B. Chen. 1994. "Avoiding conflict misses dynamically in large direct-mapped caches." In *Proc. Sixth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS'94)*, pages 158–170, San Jose CA.

S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee. 1996. *Software Synthesis from Dataflow Graphs*. Kluwer Academic Publishers.

B. Calder, C. Krintz, S. John, and T. Austin. 1998. "Cache-conscious data placement." In *Proc. Eighth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS'98)*, pages 139–149, San Jose CA.

S. Carr. 1993. *Memory-Hierarchy Management*. PhD thesis, Rice University.

S. Carr, K. S. McKinley, and C. Tseng. 1994. "Compiler optimizations for improving data locality." In *Proc. Sixth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS'94)*, pages 252–262, San Jose CA.

J. L. Hennessy and D. A. Patterson. 1996. *Computer Architecture: A Quantitative Approach, 2nd Edition*. Morgan Kaufmann Publishers, Inc.

B. L. Jacob, P. M. Chen, Seth R. Silverman, and T. N. Mudge. "An analytical model for designing storage hierarchies." *IEEE Transactions on Computers*, vol. 45, no. 10, pp. 1180–1194, October 1996.

B. L. Jacob and T. N. Mudge. 1998a. "A look at several memory-management units, TLB-refill mechanisms, and page table organizations." In *Proc. Eighth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS'98)*, pages 295–306, San Jose CA.

B. L. Jacob and T. N. Mudge. 1998b. "Virtual memory in contemporary microprocessors." *IEEE Micro*, 18(4):60–75.

B. L. Jacob and T. N. Mudge. 1998c. "Virtual memory: Issues of implementation." *IEEE Computer*, 31(6):33–43.

P. Lapsley, J. Bier, A. Shoham, and E. A. Lee. 1994. *DSP Processor Fundamentals: Architectures and Features*. Berkeley Design Technology, Inc., Berkeley CA.

E. A. Lee and D. G. Messerschmitt. 1987. "Synchronous dataflow." *Proceedings of the IEEE*, 75(9):1235–1245.

S. McFarling. 1989. "Program optimization for instruction caches." In *Proc. Third Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS'89)*, pages 183–191.