CSAIL

Computer Science and Artificial Intelligence Laboratory

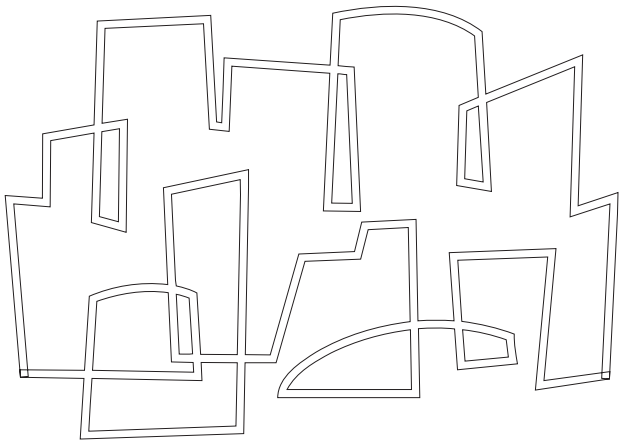
Massachusetts Institute of Technology

Application-Specific Memory Management for Embedded Systems

Prabhat Jain, Derek Chiou,
Srinivas Devadas, Larry Rudolph

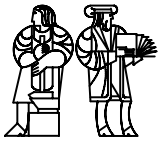
1999, November

Computation Structures Group
Memo 427



The Stata Center, 32 Vassar Street, Cambridge, Massachusetts 02139

LABORATORY FOR
COMPUTER SCIENCE



MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

Application-Specific Memory Management for Embedded Systems Using Software-Controlled Caches

Computation Structures Group Memo 427

Derek Chiou

Prabhat Jain

Srinivas Devadas

Larry Rudolph

Massachusetts Institute of Technology
Laboratory for Computer Science
545 Technology Square, Cambridge, Massachusetts
{**derek,prabhat,devadas,rudolph**}**@lcs.mit.edu**

This paper describes research performed at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for this work is provided in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-92-J-1310.

Application-Specific Memory Management for Embedded Systems Using Software-Controlled Caches

Derek Chiou Prabhat Jain Srinivas Devadas
Larry Rudolph
Massachusetts Institute of Technology

Laboratory for Computer Science
545 Technology Square, Cambridge, Massachusetts
{derek,prabhat,devadas,rudolph}@lcs.mit.edu

Abstract

We propose a methodology to improve the performance of embedded processors running data-intensive applications by allowing embedded software to manage on-chip memory on an application-specific or task-specific basis. We provide this management ability with a novel hardware mechanism, *column caching*.

Column caching provides software with the ability to dynamically partition the cache. Data can be placed within a specified set of cache "columns" to avoid conflicts with other cached items. By mapping a column-sized region of memory to its own column, column caching can also provide the same functionality as a dedicated scratchpad memory including predictability for time-critical parts of a real-time application. Column caching enables the ability to dynamically change the ratio between scratchpad size and cache size for each application, or each task within an application. Thus, software has much finer software control of on-chip memory.

We present column caching and techniques to automatically layout program data structures in a column cache; these techniques achieve significant improvements in performance because on-chip memory is utilized effectively.

1 Introduction

Due to their enormous market demand, the manufacturing of electronic systems is very cost-sensitive, and many applications (e.g., cellular phones) have stringent requirements on power consumption for the sake of portability. Manufacturers profit from integrating an *entire system* on a single integrated circuit (IC) [10] [6]. As time-to-market requirements place greater burden on designers for fast design cycles, programmable components are introduced in the system and an increasing amount of system functionality is implemented in software relative to hardware. These programmable components, called *embedded processors* or *embedded controllers*, can be general-purpose microprocessors, off-the-shelf digital signal processors (DSPs), in-house application-specific instruction-set processors (ASIPs), or microcontrollers. Systems containing programmable processors that are employed for applications other than general-purpose computing are called *embedded systems*. This paper investigates an architectural modification that can improve the efficiency and performance of embedded systems by enabling application-specific memory management via software-controlled caches.

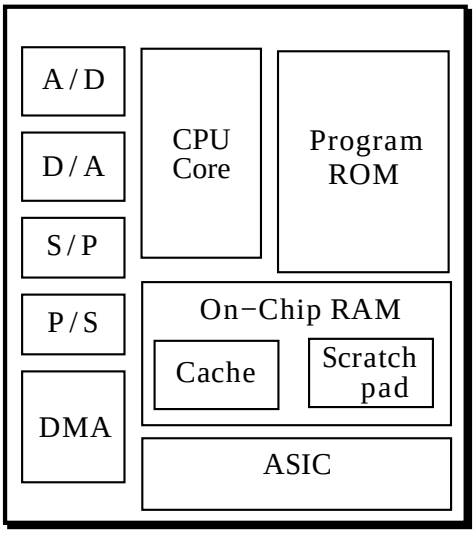


Figure 1: **A heterogeneous system-on-a-chip.**

The advantages of incorporating software components are twofold. First, whenever a software solution offers acceptable performance, it is preferred to hardware since it dramatically reduces the need for custom hardware. Only the most time-critical tasks need to be implemented in hardware. Second, since embedded processors are field-programmable, software design is more flexible than hardware design, and design errors, late specification and design changes, and product evolution can be accommodated more easily [20] [19].

As a result of the advantages of software, modern integrated systems are often composed of a heterogeneous mixture of hardware and software components. For instance, Figure 1 presents one such heterogeneous system, consisting of a digital signal processor (DSP) core, a program ROM, on-chip RAM, application-specific circuitry (ASIC), and other interface and peripheral circuitries. The on-chip RAM can be implemented as a combination of cache, scratchpad SRAM, and more recently, as embedded DRAM. Instruction and data cache are fast, on-chip memories forming an interface between the processor and off-chip memory, reducing the effective memory access time by exploiting data locality [11].

1.1 Caches and Scratchpad Memory

Caches can significantly speed up application performance by exploiting spatial and temporal locality in memory access patterns, while being transparent to the programmer or compiler. Caches are automatically filled by hardware-implemented replacement algorithms; their performance is defined by how well the replacement algorithm predicts future reference patterns.

Cache performance often degrades significantly when multiple data streams with very different locality characteristics compete for the same cache resources. Cache performance is dependent on the dynamic

memory reference stream creating unpredictability. As the memory hierarchy deepens the variance in access times increase as well.

For these reasons, caches are not prevalent in real-time embedded systems where predictability is paramount. Rather, designers use scratchpad memory, i.e., software-controlled on-chip memory located in a separate address region, for predictable execution time. Although moving data between scratchpad memory and standard memory requires explicit copies, once data is in the scratchpad memory, performance is completely predictable. Typically, the scratchpad memory is used to store critical data. Effectively utilizing scratchpad memory, however, adds software complexity. A data structure that does not fit in the scratchpad, e.g., a large matrix, either cannot be assigned to the scratchpad or must be subdivided and swapped in and out of the scratchpad, complicating the application code. Caches and scratchpad memory each have their advantages and disadvantages.

Having both a cache and a scratchpad memory in an embedded processor provides both predictability and high performance for a wide range of applications. Data accessed throughout the execution of the application or with high temporal locality should reside in scratchpad memory. Data structures that are larger than the scratchpad memory size and that have temporal or spatial locality are best assigned to caches.

During the *design phase* of an embedded system, on-chip memory organization can be tailored to the requirements of a given application. For example, the work of Panda *et al* [18] shows that significant performance gains result from effectively partitioning on-chip memory into scratchpad RAM and cache. Their data shows that the optimal partition varies for different procedures in the same application. Thus, it is often the caset that a static partitioning performs suboptimally across a range of applications.

Fisher argues, on the basis of product development cycles, that processor choices in embedded systems are usually bound six months to a year ahead of the first shipment, and that the software can change significantly within that period [8]. Since it is hard to know the application one is customizing for, specializing memory for a specific use may not be viable.

1.2 Our Work

One way of achieving high performance over a range of applications is to allow for some form of hardware configurability that can be controlled in a dynamic manner. Of course, this configurability will come at an additional hardware cost, and is only worthwhile if the gains provided by configurability exceed the hardware cost.

We propose a methodology to improve the performance of embedded processors running data-intensive applications by allowing embedded software to *manage on-chip memory on an application-specific or task-*

specific basis. We accomplish task-specific management using a new hardware mechanism we term column caching [4]. *Column caching* provides software control as to where items are stored in a cache. In our reference implementation each column can be viewed as one "way" or bank of an n-way set-associative cache. Program data structures are placed within a specified set of cache columns so as to avoid conflicts with other cached items. By exclusively allocating a region of memory to an equal-sized region of cache, columns may also be used as scratchpad memory.

Column caches provide a mechanism by which on-chip memory can be optimally partitioned into scratch-pad memory and cache for each application, or each task within an application. Column caching can both provide predictability for time-critical parts of a real-time application and improve performance. Determining the appropriate mapping of program data structures to cache or scratchpad RAM, or more generally, to different columns in the column cache requires program analysis. We present techniques to automatically map program data structures to a column cache; these techniques, when implemented in the front-end of a compiler, can achieve significant improvements in performance for embedded software kernels because on-chip memory is utilized more effectively.

1.3 Organization of the paper

The remainder of the paper is organized as follows. We first describe column caching in Section 2. We describe our data layout algorithm to exploit the control provided by the column caching mechanism in Section 3. Experimental results are presented in Section 4, and related work is discussed in Section 5. Conclusions and ongoing work are the subject of Section 6.

2 Column Caching

Standard caches treat all cached memory locations the same. In a set-assocative cache the same replacement algorithm is applied to all memory accesses and the replaced cache-line is always selected from the entire set. Thus, the cache is seen as a single entity to the hardware.

Column caching allows software to map specific data[1] to specific regions of the cache. Column caching conceptually partitions the cache into *columns* that can then be assigned to cache data that satisfies some criteria. Careful mapping can reduce or eliminate some replacement errors, resulting in improved performance. In addition, column caching enables a cache to emulate scratchpad memory, separate spatial/temporal

[1] We will use the term *data* to mean either instructions or data.

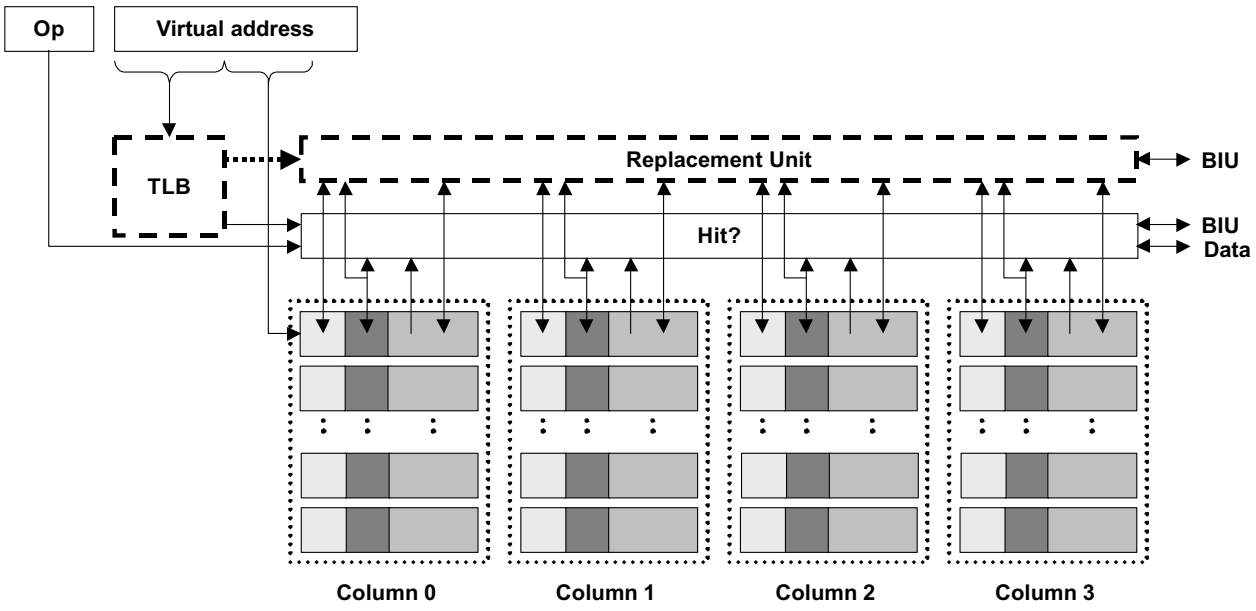


Figure 2: **Basic Column Caching.** Three modifications to a set-associative cache, denoted by dotted lines in the figure, are necessary: (i) augmented TLB to hold mapping information, (ii) modified replacement unit that uses mapping information and (iii) a path between the TLB and the replacement unit that carries that information.

caches, a separate prefetch buffer, separate write buffers and other traditional, statically-partitioned structures within the general cache.

2.1 Implementation

The simplest implementation of column caching is derived from a set-assocative cache where lower-order bits are used to select a *set* of cache-lines which are then associatively searched for the desired data. If the data is not found (a cache miss), the replacement algorithm selects a cache-line from the selected set.

During lookup, a column cache behaves exactly as a standard set-assocative cache and thus incurs no performance penalty on a cache hit. Rather than allowing the replacement algorithm to always select from any cache-line in the set, however, column caching provides the ability to restrict the replacement algorithm to certain *columns*. Each column is one "way", or bank, of the n-way set-associative cache (Figure 2). A bit vector specifying the permissible set of columns is generated and passed to the replacement unit.

By mapping all regions of memory across all columns, the cache becomes a normal set-associative cache. By aggregating columns into partitions, we can provide set-associativity within partitions as well as increase the size of partitions. Since every cache-line in the set is searched during every access, repartitioning is graceful; if data is moved from one column to another (but always in the same set), the associative search will still find the data in the new location. A memory location can be cached in one column during one

cycle, then re-mapped to another column on the next cycle. The cached data will not move to the new column instantaneously, but will remain in the old column until it is replaced. Once removed from the cache, it will be cached in a column to which it is mapped the next time it is accessed.

Column caching is implemented via three small modifications to a set-associative cache (Figure 2). The TLB must be modified to store the mapping information. The replacement unit must be modified to respect TLB-generated restrictions of replacement cache-line selection. A path to carry the mapping information from the TLB to the replacement unit must be provided. Similar control over the cache already exists for uncached data, since the cached/uncached bit resides in the TLB.

2.2 Partitioning and Repartitioning

Implementation is greatly simplified if the minimum mapping granularity is a page, since existing virtual memory translation mechanisms including the ubiquitous translation-look-aside-buffers (TLB) can be used to store mapping information that will be passed to the replacement unit. TLB's, accessed every memory reference, are designed to be fast in order to minimize physical cache access time. Partitioning is supported by simply adding column caching mapping entries to the TLB data structures and providing a data path from those entries to the modified replacement unit. Therefore, in order to remap pages to columns, access to the page table entries is required.

Mapping a page to a cache partition represented by a bit vector is a two phase process. Pages are mapped to a *tint* rather than to a bit vector directly. A tint is a virtual grouping of address spaces. For example, an entire streaming data structure could be mapped to a single tint, or all streaming data structures could be mapped to a single tint, or just the first page of several data structures could be mapped to a single tint. Tints are independently mapped to a set of columns, represented by a bit vector; such mappings can be changed quickly. Thus, tints, rather than bit vectors, are stored in page table entries.

The tint level-of-indirection is introduced (i) to isolate the user from machine-specific information such as the number of columns or the number of levels of the memory hierarchy and (ii) to make re-mapping easier. The second reason is illustrated in Figure 3. If a region's tint is changed (re-tinted), each page table entry of that region needs to be updated and any corresponding TLB's either updated or flushed. Re-tinting should occur very infrequently compared to remapping tints to bit-vectors.

2.3 Using Columns As Scratchpad Memory

Column caching can emulate scratchpad memory within the cache by dedicating a region of cache equal in size to a region of memory. No other memory regions are mapped to the same region of cache. Since there

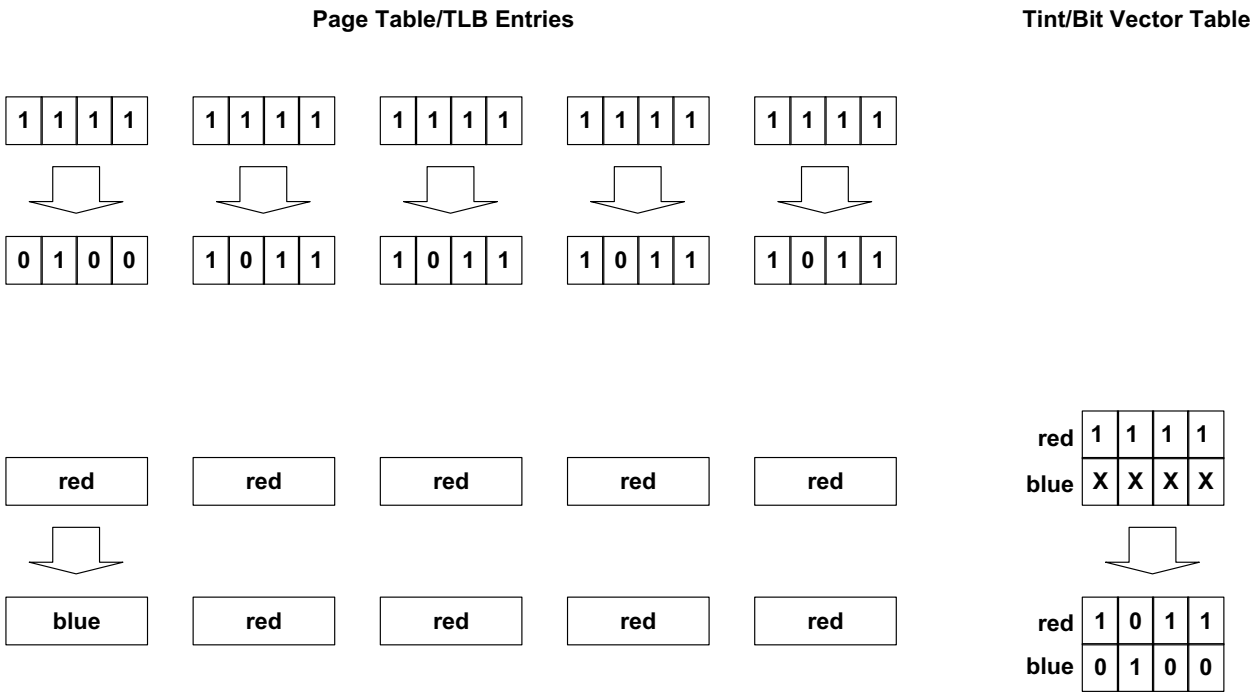


Figure 3: **Usefulness of Tints.** An example that demonstrates the advantage of storing tints instead of bit vectors in the page table entries. In the first example, where we store raw bit vectors in page table entries, in order to remap page 0 to use its own column and the rest of the pages to use the remaining columns, we need to change all page table entries. In the second example, using tints, only one tint and two tint/bit-vector table entries need to be changed. All pages start with the default tint, red. In order to give one page its own column, that page's tint is changed to blue. Tint blue corresponds to a bit vector that specifies that page to be cached in the second column. Tint red's bit vector is changed to remove the second column as a possible replacement column. The TLB entries for all former tint red pages must be flushed or modified in place to reflect the new bit vector.

is a one-to-one mapping, once the data is brought into the cache it will remain there. In order to guarantee performance, software can perform a load on all cache-lines of data when remapping as is required with a dedicated SRAM. That memory region then behaves like a scratchpad memory.

2.4 Impact of Column Caching on Clock Cycle

The modifications required for column caching are limited to the cache replacement unit which is not on the critical path. In realistic systems, data requested from L1 cache to main memory takes at least three cycles to return. One cycle is needed to pipeline the request to main memory, one cycle to read the data, and one cycle to pipeline the reply. Most systems take at least several more cycles from initiation to completion of a memory request since there are generally more pipeline stages. The exact replacement cache-line does not need to be decided until the data returns, giving the replacement algorithm at least 3 to 5 cycles to make a decision. Though some redesign of the TLB and replacement unit are necessary to incorporate column caching information and replacement algorithms, such modifications are very unlikely to impact cycle time since there is so much time available to make a decision.

3 Data Layout Algorithm

An appropriate data layout is required to maximize the advantage of enhanced cache functionality. In particular, program data structures have to be assigned to columns, or sets of columns. This assignment can be static, i.e., made once prior to running the program, or can vary dynamically during program execution. We will describe techniques for static data layout in this paper, and briefly outline issues involved with dynamic data layout.

3.1 Static Data Layout

We describe our algorithm to assign program variables to columns in a column cache. The cache C is composed of a set of k columns $\{c_1, \ldots, c_k\}$. We assume that each column c_k is of the same size, i.e., $|c_k| = S$.

1. Variables that are to be explicitly assign to columns are first identified. Heavily accessed scalar variables are denoted as $s_i,\ 1 \leq i \leq M$ and array variables as $v_i,\ 1 \leq i \leq N$, where the size of v_i is denoted $|v_i|$. If a variable v is larger than the size of a column S, even if v is exclusively assigned, we cannot treat it as scratchpad memory because elements of v may replace other elements of v. Thus, v

is split into separate subarrays, each of which can fit into a column. Similarly, a set of variables can be aggregated into a single variable which is assigned to a column c_j such that $\sum_{i=1}^{p} |v_i| \leq S$.

2. We construct a complete undirected graph $G(V, E, W)$ where each vertex v_i corresponds to a program array variable. Given two vertices v_i and v_j, we compute a cost $w(v_i, v_j)$ for each edge using the algorithm below. This weight $w(v_i, v_j)$ represents the cost of placing v_i and v_j in the same column. These weights are computed using the algorithm described in Section 3.1.1.

3. We determine a mapping of vertices in G, namely $M : V \rightarrow C$ where C is the set of columns in the column cache. Each variable is assigned to exactly one column $c_j \in C$. [2] We do this such that the total weight W given below is minimized.

$$W = \sum_{j=1}^{|E|} w(e_j) * b_j$$

where $e_j =< v_{j1}, v_{j2} >$ and $b_j = 1$ if $C(v_{j1}) = C(v_{j2})$ else $b_j = 0$. This cost function simply models the cost (relating to misses due to conflicts) of placing two variables into the same column. Thus, this problem is similar to graph-coloring, where colors correspond to columns.[3] If the graph is k-colorable, where k is the number of columns, then we will have a minimum-cost solution with $W = 0$. However, if the graph is not k-colorable, then we still have to color the vertices of the graph so W is minimized. It is clear that our problem is at least as hard as graph k-colorability which is NP-complete [9]. We use a heuristic algorithm to solve our problem, described in Section 3.1.2.

3.1.1 Determining Weights

We describe our method of determining the weights of edges in graph G. Our goal in determining weights is *not* to measure miss rate, but to quantify the number of potential conflicts that arise when two variables are assigned to the same column. We wish the computed weights to be accurate in a relative, rather than absolute manner. That is, if variables v_i and v_j have an edge with high weight between them, and if v_i and v_k have an edge with low weight, then placing v_i and v_k in the same column should produce fewer conflicts than placing v_i and v_j in the same column.

We have two ways to determining weights: (i) a profile-based method and (ii) a faster, approximate program analysis method.

[2] The functionality of a column cache allows us to assign a variable to any subset of columns in C. But for the purposes of this paper, we will restrict ourselves to assigning variables to a single column.

[3] Note that while G is a complete graph, prior to coloring, we will delete all zero-weight edges.

In our profile-based method, we run the program on a representative data set to obtain a sequence of variable accesses. This sequence can be used to determine the heavily-accessed scalars (Step 1 above), as well as the number of accesses of each array variable.

The life-time of a variable is defined as the period between its definition and last use [1]. Arrays with disjoint life-times can be placed in the same column without fear of conflict. We therefore obtain the life-time distribution of each array variable from the address sequence, i.e., for each array variable v we determine $I(v_i) = [first_i,\ last_i]$.

Given two variables v_i and v_j, we use $I(v_i)$ and $I(v_j)$ to compute the weight of the edge $w(v_i, v_j)$. If $I(v_i) \cap I(v_j) = \phi$, then $w(v_i, v_j)$ is 0. Else, we find the intersection interval where both variables are live, namely $\Pi_{i,j} = [MAX(first_i, first_j),\ MIN(last_i, last_j)]$. During this interval $\Pi_{i,j}$, we determine the number of accesses to variable v_i and variable v_j, call them n_i^j and n_j^i, respectively. Note that it is possible that neither of these variables is actually accessed during $\Pi_{i,j}$. We assign the weight to be $w(v_i, v_j) = MIN(n_i^j, n_j^i)$. This measures the number of accesses that potentially conflict if v_i and v_j are assigned to the same cache column.

The program analysis method operates on the intermediate form (IF) representation of the program used in compilers. Using the IF representation, we determine approximate life-times of each array variable. For each variable, we determine the number of accesses by estimating loop iteration counts and the the probability of taking branches.

3.1.2 Column Assignment

The heuristic algorithm for column assignment we use is described here. Some recent work in exact graph coloring has shown that if graphs are 1-perfect then one can find a minimum coloring within reasonable time [5]. Therefore, given a graph G, we first delete all the zero-weight edges in G. We then find a minimum coloring of the graph using the algorithm proposed by Coudert in [5]. If the number of colors required is less than or equal to k, we have found a solution to our problem with $W = 0$ that is clearly optimal. We simply assign the variables to columns based on the computed minimum coloring. This is very rare, since there are typically many more vertices in the graph than k, and our graphs have $O(V^2)$ edges.

If the number of colors required is more than k, we do the following. We find the minimum-weight edge in G and merge the vertices that are connected by this edge. This results in a smaller graph with one less vertex. We run exact minimum graph coloring on this graph. Again, we check to see the number of colors required. We stop when the number of colors required is less than or equal to k, and assign columns to vertices by the coloring. Any merged vertices are assigned to the same column.

3.1.3 Forcing Variables to Scratchpad Memory

If a variable v is heavily accessed through a long interval of time while other variables are also being accessed, then the weights of edges connecting v to these other variables/vertices will all have high weight. The graph coloring will tend to place v into its own column. If $|v| \leq S$ then v has effectively been placed in scratchpad memory.

However, in some cases it may be beneficial to force certain variables into scratchpad memory for predictability reasons. It is easy to do this simply by pre-assigning certain variables to p columns which serve as scratchpad memory, and reducing the number of columns from k to $k - p$.

3.2 Dynamic Data Layout

Since column mappings can be changed almost instantaneously, one can perform re-assignments at any point within an application, with minor overheads. For example, we can use the static data layout algorithm on individual procedures or sub-procedures rather than the entire application program, and if re-assignment of variables to columns is warranted, i.e., there is a significant performance benefit to re-assigning particular variables, we will change the column mapping prior to executing the procedure.

Note that if procedures have disjoint sets of variables, there is no need for re-assignment, since all the data structures can be statically mapped to columns. If procedures share variables, and the access patterns corresponding to these shared variables change from procedure to procedure within an application, it is worthwhile to consider remapping.

4 Experiments

This section presents experimental verification of the benefits of column caching both for a single application as well as for a set of applications executing in a multi-tasking fashion.

4.1 Scratchpad Memory Versus Cache

We consider three main routines of an MPEG application, `dequant`, `plus` and `idct`, as an embedded benchmark (we follow the example of Panda [18]). For each of these routines, the amount of memory is fixed at 2KB and the ratio between cache and scratchpad memory is varied. There are four columns in this cache. At one extreme, all four columns are used as a scratchpad, and at the other extreme, all four columns used as a 4-way set-associative cache. For each routine, its performance is measured as a function of the

cache size. For each memory partition, the data layout algorithm was used to determine the mapping of variables to columns.

Figures 4(a) through 4(c) show the results for each of the three routines, and Figure 4(d) is the combined result. For two routines, `dequant` and `plus`, optimum performance is achieved simply by assigning all data to scratchpad memory since all the heavily accessed data fits into the scratchpad, and cold misses are avoided. When the scratchpad memory size is 0, performance degrades due to cold misses in the cache. The opposite occurs in `idct` where its data structures are larger than 2KB and cannot make effective use the scratchpad. Appropriate assignment to different cache columns, yields superior performance.

These three routines illustrate the important point that the optimum partition of scratchpad and cache varies from procedure to procedure even within the same application. If a static partition is chosen, the performance that can be obtained is shown in Figure 4(d) for different partitions. Column caching enables dynamic partitioning, by allowing the remapping of variables to columns, and the performance improvement that can be obtained by a column cache is shown in Figure 4(d).

4.2 Multitasking

Column caching enables predictable performance within a multitasking environment where multiple jobs are executing. We consider three compression (gzip) jobs simultaneously executing on one processor and each having access to the cache. To understand what is happening, we only present the performance measurement of a one job in this mixture. We present the results in terms of clocks per instruction (CPI) which is inversely corrolated with performance – a lower CPI means higher performance. Figure5, shows the variation in job A's CPI when the time quantum is varied. We present results for both a standard cache and a mapped column cache. The two sets of plots correspond to different sized caches.

Consider a memory of 16K (the topmost curve). Each point in this plot was generated by choosing a time quantum, and performing a round-robin schedule of the three jobs, A, B, and C. Each job gets to use the entire cache when it is running. There is a significant difference in the CPI for job A, as the time quantum varies. This variation is caused mainly by the cache hit rate for job A being affected by intervening cache accesses due to jobs B and C. The number of such accesses is dependent on the time quantum. However, if job A is assigned to its own columns in a column cache, with job B and job C sharing the rest of the cache, then the CPI of job A is significantly less sensitive to the time quantum, as shown by the curve immediately below (second from the top). Job A was considered critical, and it was exclusively assigned a large fraction of the cache, hence the hit rate for job A is higher. Therefore, the CPI is significantly smaller for small time quanta. Of course when the time quantum is really large, we effectively have batch processing and the CPI's

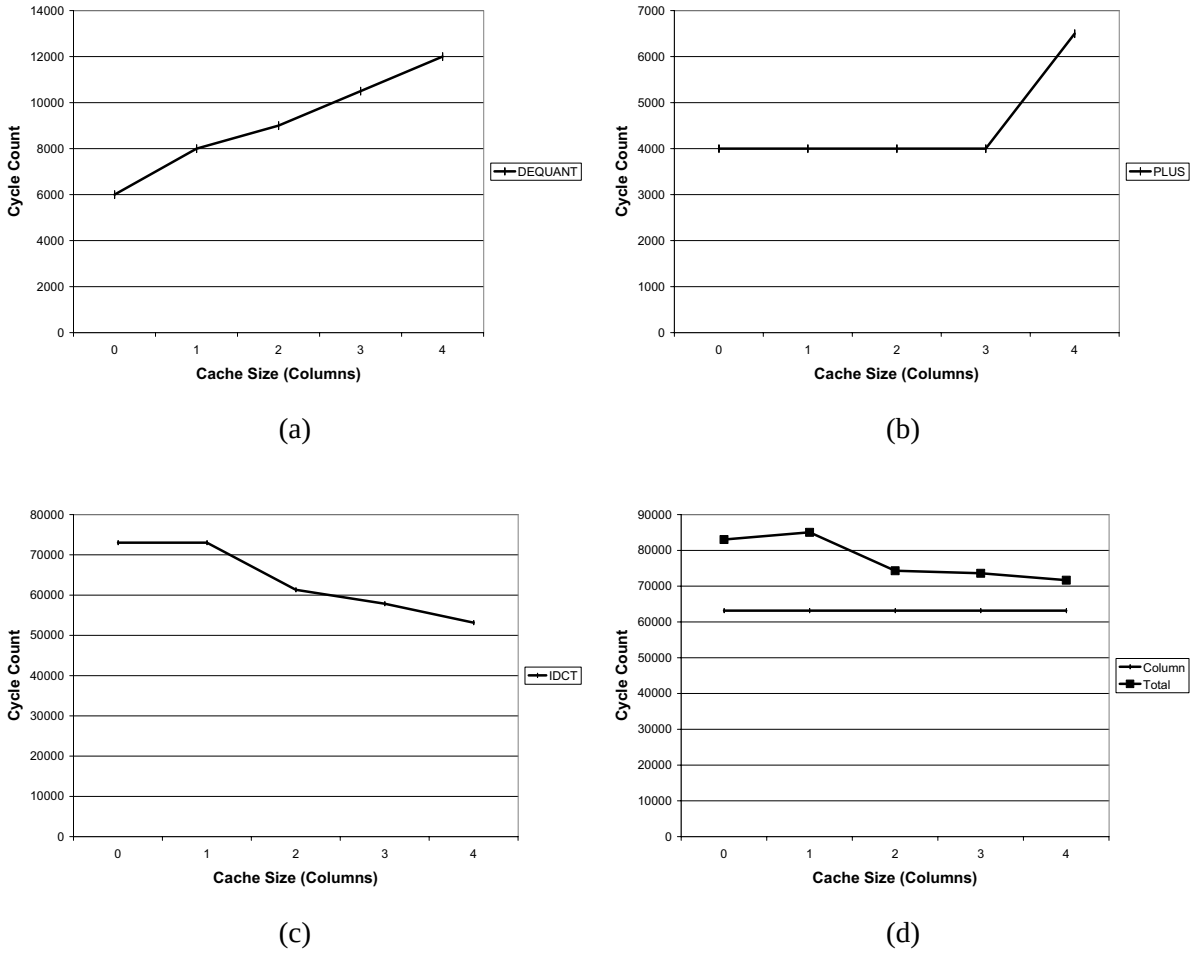


Figure 4: (a) Performance of `Dequant` routine with varying scratchpad/cache partitions (b) Performance of `Plus` routine (c) Performance of `Idct` routine (d) Performance of overall application, and performance using a column cache.

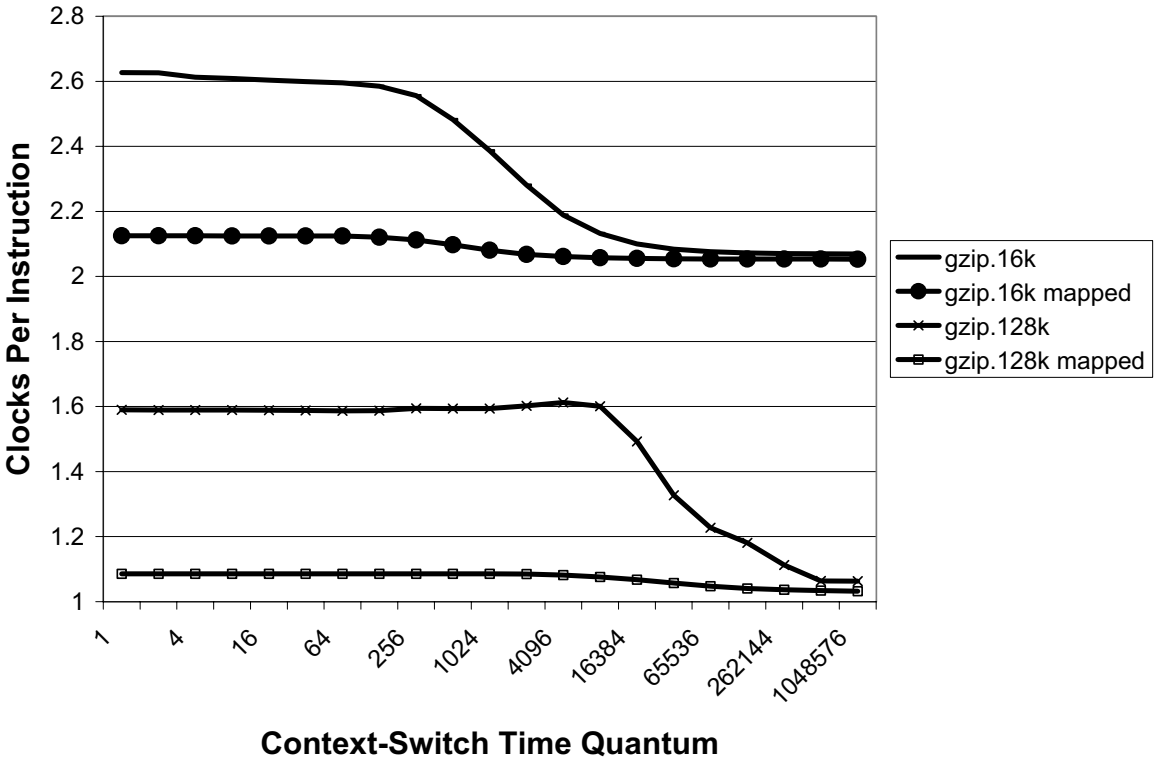


Figure 5: Column caching can significantly improve performance as well as reduce performance variation with varying time quanta.

are virtually the same for the top two plots.

If a smaller fraction of the cache is assigned to job A, job A's performance would degrade; however, *the performance variation would not increase*. Similarly if the cache size is increased, the CPI's become smaller, but the performance variation for the mapped column cache is very small, as shown in the bottom two curves.

One may argue that the time quantum could be fixed for predictability, but in reality due to interrupts and exceptions the effective time quantum can vary significantly during the time that a job is running simultaneously with other jobs. Thus, column caching can improve performance of a critical job as well as significantly reduce performance variation in the presence of interrupts or varying time quanta.

5 Related Work

5.1 Cache Mechanisms

The idea of statically-partitioned caches has been around for a long time. The most common example are separate instruction and data caches. Some existing and proposed architectures support a pair of caches, one

for spatial locality and one for temporal locality [21, 23, 12, 2, 15, 7]. These designs statically separate the two caches in hardware, generally wasting resources since the partition is rarely exactly correct.

Sun Microsystems Corporation holds a patent on a mechanism [17] very similar to column caching that allows partitioning of a cache between processes at cache column granularity. As part of a process state, a bit mask is specified that indicates which columns can be replaced by that process. The Sun technique is limited to partitioning between processes and does not address other criteria such as memory address ranges, memory operations and memory operation addresses.

Page coloring refers to intelligent mapping of virtual pages to physical pages to reduce conflicts in a direct-mapped cache and thus offers a limited sub-set of column caching abilities. Page coloring does not require any special hardware support beyond the address translation found in virtually all general-purpose processors. Column caching provides much of the same functionality as page coloring, but eliminates the limitations of page coloring. For example, page coloring requires a memory copy to remap a region of memory to a new region of the cache, while column caching can do common remappings almost instantanously. Column caching also works well with set-associative caches, where page coloring potentially wastes a significant amount of space.

The Impulse [3] project proposes something very similar to remapping addresses, but within the memory controller rather than the cache. By changing mappings within the memory controller, non-contiguous data can be packed into a contiguous region of memory before being sent over the bus, thus saving bandwidth. The modifications were proposed to be made within the memory controller in order to avoid any changes the the processor. A major disadvantage of such a scheme is that the software must be aware of the remapping, which is difficult and thus would make such remappings infrequent. In addition, memory controllers are not necessary less complex than the processors they support, since they must deal with network interfaces, a variety of peripheral interfaces, and multiple processors.

5.2 Memory Exploration in Embedded Systems

Cache memory issues have been studied in the context of embedded systems. McFarling presents techniques of code placement in main memory to maximize instruction cache hit ratio [16, 25]. A model for partitioning an instruction cache among multiple processes has been presented [14].

Some efforts have focused on the possibility of power reduction during memory accesses. Instruction scheduling for power reduction in instruction caches has also been addressed [24]. The problem of mapping set data structures used in networking applications into memory so as to reduce power consumption has also been studied [26].

A system-level memory exploration technique targeting ATM applications is presented in [22]. A simulation-based technique for slecting a processor and required instruction and data caches is presented in [13].

The work closest to our own is the work of Panda, Dutt and Nicolau who present techniques for partitioning on-chip memory into scratchpad memory and cache [18]. The presented algorithm assumes a fixed amount of scratchpad memory and a fixed-size cache, identifies critical variables and assigns them to scratchpad memory. The algorithm can be run repeatedly to find the optimum performance point. Our algorithm targets a column cache and determines a mapping of variables to columns which may end up being used as scratchpad memory or cache or both.

6 Conclusions

The work described in this paper represents a confluence of two observations. The first observation is that given heterogeneous applications with data streams that have significant variation in their locality properties, it is worthwhile to provide finer software control of the cache so the cache can be used more efficiently. To this end, we have proposed a column caching mechanism that enables cache partitioning so data with different locality characteristics can be isolated for improved performance. The second observation is that columns can emulate scratchpad memory which is used extensively to improve predictability in embedded systems. A significant benefit of columns is that through the execution of a program, the data stored in columns can explicitly managed as in a scratchpad or can be implicitly managed as in a cache.

The two observations above led us to develop a data layout algorithm for embedded software that utilizes the features of a column cache. This algorithm assigns program data structures, i.e., memory addresses to columns in a column cache. A column, and therefore part of a cache, may store critical data at one point in program execution and behave like a scratchpad, and may store non-critical data and behave like an ordinary cache at other points. This configurability results in improved performance, as evidenced by our experimental data.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers Principles, Techniques and Tools*. Addison-Wesley, 1986.

[2] Krste Asanovic. *Vector Microprocessors*. PhD thesis, University of California at Berkeley, May 1998.

[3] J.B. Carter, W.C. Hsieh, L.B. Stoller, M.R. Swanson, L. Zhang, E.L. Brunvand, A. Davis, C.-C. Kuo, R. Kuramkote, M.A. Parker, L. Schaelicke, and T. Tateyama. Impulse: Building a Smarter Memory

Controller. In *Fifth International Symposium on High Performance Computer Architecture*, pages 70–79, January 1999.

[4] Derek T. Chiou. *Extending the Reach of Microprocessors: Column and Curious Caching*. PhD thesis, Department of EECS, MIT, Cambridge, MA, September 1999.

[5] O. Coudert. Exact Coloring of Real-Life Graphs is Easy. In *Proceedings of the 34^{th} Design Automation Conference*, pages 121–126, June 1997.

[6] F. Depuydt. *Register Optimization and Scheduling for Real-Time Digital Signal Processing Architectures*. PhD thesis, Katholieke Universiteit Leuven, October 1993.

[7] Greg Faanes. A CMOS Vector Processor with a Custom Streaming Cache. In *Hot Chips 10*, August 1998.

[8] J. Fisher. Customized Instruction-Sets for Embedded Processors. In *Proceedings of the 36^{th} Design Automation Conference*, pages 253–257, June 1999.

[9] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. W. H. Freeman and Company, 1979.

[10] G. Goossens, F. Catthoor, D. Lanneer, and H. De Man. Integration of Signal Processing Systems on Heterogeneous IC Architectures. In *Proceedings of the 6th International Workshop on High-Level Synthesis*, pages 16–26, November 1992.

[11] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufman, 1990.

[12] Intel. *Intel StrongARM SA-1100 Microprocessor*, April 1999.

[13] D. Kirovski, C. Lee, M. Potkonjak, and W. Mangione-Smith. Application-Driven Synthesis of Core-Based Systems. In *Proceedings of the International Conference on Computer-Aided Design*, pages 104–107, November 1997.

[14] Y. Li and W. Wolf. A Task-Level Hierarchical Memory Model for System Synthesis of Multiprocessors. In *Proceedings of the 34^{th} Design Automation Conference*, pages 153–156, June 1997.

[15] Bill Lynch and Gary Lauterbach. UltraSPARC III: A 600 MHz 64-bit Superscalar Processor for 1000-Way Scalable Systems. In *Hot Chips 10*, 1998.

[16] S. McFarling. Program Optimization for Instruction Caches. In *Proceedings of the 3^{d} Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, pages 183–191, April 1989.

[17] Basem Nayfeh and Yousef A Khalidi. Us patent 5584014: Apparatus and method to preserve data in a set associative memory device, December 1996.

[18] P. R. Panda, N. Dutt, and A. Nicolau. *Memory Issues in Embedded Systems-on-Chip: Optimizations and Exploration*. Kluwer Academic Publishers, 1999.

[19] P. G. Paulin, C. Liem, T. C. May, and S. Sutarwala. DSP Design Tool Requirements for Embedded Systems: A Telecommunications Industrial Perspective. *Journal of VLSI Signal Processing*, 9(1/2):23–47, January 1995.

[20] J. Van Praet, G. Goossens, D. Lanneer, and H. De Man. Instruction Set Definition and Instruction Selection for ASIPs. In *Proceedings of the 7th IEEE/ACM International Symposium on High-Level Synthesis*, May 1994.

[21] F.J. Sánchez, A. González, and M. Valero. Software Management of Selective and Dual Data Caches. In *IEEE Computer Society Technical Committee on Computer Architecture: Special Issue on Distributed Shared Memory and Related Issues*, pages 3–10, March 1997.

[22] P. Slock, S. Wuytack, F. Catthoor, and G. de Jong. Fast and Extensive System-Level Memory Exploration and ATM Applications. In *Proceedings of 1995 International Symposium on System Synthesis*, pages 74–81, 1997.

[23] M. Tomasko, S. Hadjiyiannis, and W.A. Najjar. Experimental Evaluation of Array Caches. In *IEEE Computer Society Technical Committee on Computer Architecture: Special Issue on Distributed Shared Memory and Related Issues*, pages 11–16, March 1997.

[24] H. Tomiyama, T. Ishihara, A. Inoue, and H. Yasuura. Instruction Scheduling for Power Reduction in Processor-Based System Design. In *Design, Automation, and Test in Europe*, February 1998.

[25] H. Tomiyama and H. Yasuura. Code Placement Techniques for Cache Miss Rate Reduction. *ACM Transactions on Design Automation of Electronic Systems*, 2(4):410–429, October 1997.

[26] S. Wuytack, F. Catthoor, and H. De Man. Transforming Set Data Types to Power Optimal Data Structures. In *Proceedings of the Int'l Symposium on Low Power Design*, pages 51–56, April 1995.