# Data Cache Sizing for Embedded Processor Applications *

Preeti Ranjan Panda Nikil D. Dutt Alexandru Nicolau

Department of Information and Computer Science
University of California, Irvine, CA 92697-3425, USA

## Abstract

*We present a technique for determining the best data cache size required for a given memory-intensive application. A careful memory and cache line assignment strategy based on the analysis of the array access patterns effects a significant reduction in the required data cache size, with no negative impact on the performance, thereby freeing vital on-chip silicon area for other hardware resources. Experiments on several benchmark kernels performed on LSI Logic's CW4001 embedded processor simulator confirm the soundness of our cache sizing and memory assignment strategy and the accuracy of our analytical predictions.*

## 1. Introduction

The architectural flexibility offered by embedded processor-based systems makes the code generation process much more complex than traditional compilation. The processor now forms only part of the die: the system designer thus has to decide what other hardware components (e.g., memory, co-processors, etc.) will comprise the rest of the on-chip silicon area, based on an analysis of the application. For instance, if an analysis of the application reveals that the data cache hit ratio is not likely to improve for cache sizes larger than 1 KByte, the information can be utilized to allocate expensive on-chip silicon area to other hardware resources, instead of an unnecessarily large cache. When coupled with an aggressive compiler that exploits the knowledge of this optimal configuration, the impact on the overall design is significant. Given an embedded application program and a cache line size, we determine the best data cache size in cases where the application lends itself to an exact analysis. When an exact analysis is not possible (e.g., applications with input data-dependent memory reference patterns), we generate a graph plotting the expected variation of cache performance with size, for the given application, allowing a system designer to rapidly explore the performance impact of different cache configurations.

*This work was partially supported by grants from ARPA (MDA904-96-C-1472), NSF(CDA-9422095), and a UCI Dissertation Fellowship.

## 2. Data Cache Sizing

Consider a direct-mapped cache with two words per line, executing a section of code implementing matrix addition shown below.

```
int a[10][10], b[10][10], c[10][10]
...
for i = 0 to 9
   for j = 0 to 9
      c[i][j] = a[i][j] + b[i][j]
```

In any given iteration, a maximum of three distinct cache lines are accessed in the inner loop, one corresponding to each array. We can use a data layout strategy [3] which ensures that $a[i][j], b[i][j]$, and $c[i][j]$ always map to *consecutive* cache lines thereby achieving the minimum cache size of 4 lines. For example, if we map $a[0][0] \ldots a[9][9]$ to memory locations $0 \ldots 99$; $b[0][0] \ldots b[9][9]$ to locations $114 \ldots 213$; and $c[0][0] \ldots c[9][9]$ to locations $228 \ldots 327$, we observe accesses to consecutive cache lines for the first four iterations. Clearly, the absence of conflicts is also ensured in future iterations by virtue of the regular access patterns in the code. Further, the cache size (which is a power of 2) cannot be reduced to less than 4 without affecting the performance adversely. A larger cache does not improve performance due to the lack of temporal locality in the code.

### 2.1. Arrays with Compatible Access Patterns

We call two arrays accesses in a loop *compatible* if their index expressions differ by a constant (i.e., independent of loop variables). This property holds for a large variety of typical array access patterns. For example, a pair of accesses to $(A[i], A[i+2])$ and $(A[2i], B[2i+3])$ in the same loop satisfy this property.

If all accesses in a loop are compatible, then we can use a suitable data layout in memory to avoid cache conflicts completely. Consider the code fragment in Figure 1(a), to be mapped into a cache with line size = 4 words. The regions of the arrays referenced in one iteration are shown shaded in

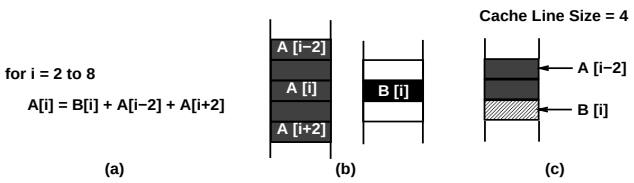


**Figure 1. (a) Example code (b) Shaded region involved in one iteration (c) Cache mapping**

Figure 1(b). A memory assignment of arrays $A$ and $B$ that avoids conflicts should ensure that the shaded regions of $A$ and $B$ never map into the same cache line. We note that the five consecutive words in $A$ can occupy, in the worst case, a maximum of two lines in this cache with four words per line. Similarly, the one word from $B$ can occupy one line. Thus, if we adjust the distance between $A[i-2]$ (the earliest $A$-word accessed in iteration $i$) and $B[i]$ (the earliest $B$-word accessed in this iteration), so that they are two lines apart when mapped into cache, we can avoid cache conflicts during loop execution.The nearest power of 2 greater than the total number of cache lines computed above would be the optimum cache size for the loop.

## 2.2. Arrays Conflicting with Scalars

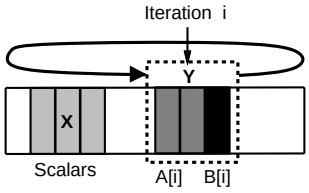


**Figure 2. Cache conflict between scalars and arrays**

Scalars accessed in loops are typically stored in the register file, but if the number of scalars exceeds the available registers, they have to be stored in main memory, and consequently, accessed through the cache. In such a scenario, conflicts between arrays and scalars in memory are inevitable, since scalars are mapped to a fixed memory location (hence, a fixed cache location), whereas the accessed array elements map to different cache locations in different iterations. Figure 2 shows the cache memory where scalars in a loop map to region $X$ of the cache, whereas elements accessed from arrays $A$ and $B$ map to different parts of the cache in different iterations. Conflicts between scalars and arrays occur whenever region $X$ (which is fixed) and $Y$ (which is moving) intersect in the cache. An estimate of the cache conflicts occurring in one loop iteration between scalars and arrays is given by: $\frac{1}{C} \cdot (M \cdot n_{sc} + M_{sc} \cdot n_a)$, where $M_{sc}$ and $M$ are the number of cache lines spanned by scalars and arrays respectively, $n_{sc}$ and $n_a$ are the number of accesses to scalars and arrays respectively, and $C$ is the number of lines in the cache. Details are described in [1].

## 2.3. Arrays with incompatible access patterns

When multiple arrays are accessed with incompatible access patterns in a loop, it is difficult to formalize a strategy to avoid conflicts altogether. To compute an estimate of the number of cache conflicts in this case, we assume a uniform distribution of the memory accesses in the cache. This is a good approximation when array access patterns are incompatible, and was verified by our experiments. We first divide the arrays into groups $S_1 \ldots S_g$, with each group $S_j$ consisting of arrays with compatible accesses, and occupying a total of $M_{S_j}$ cache lines in one iteration. The probability of the arrays interfering with any of the $n_{sc}$ scalars in cache is $M/C$ (where $M = \sum_j M_{S_j}$) , so the expected number of scalar misses is: $(M/C)n_{sc}$. Similarly, the probability that an access to any element in group $S_j$ results in a miss, is the probability that it conflicts with any scalar, or any of the other groups. Thus, it could conflict if it were to map to any of the $M_{sc} + (M - M_{S_j})$ locations occupied by the scalars and remaining arrays. In other words, the probability of a conflict miss for group $S_j = (M_{sc} + M - M_{S_j})/C$. Thus, the expected number of misses in one iteration is:

$$\#\text{Conflicts} = \frac{1}{C}\left(M \cdot n_{sc} + \sum_{j=1}^{g}(M - M_{S_j} + M_{sc})n_j\right)$$

## 3. Conclusions

In embedded systems based on microprocessor cores, architectural parameters such as on-chip memory can be tailored to the specific application that is being designed. We described a technique to predict the best data cache size for a given application analytically. Our experiments on several benchmark kernels performed on the CW4001 embedded processor core simulator, predicted optimal cache size for the applications where the prediction is possible. For the cases where an optimal cache size does not exist, our prediction of hit ratio closely follows the actual hit ratios, and permits the designer to select a good cache size for the application. The cache conflict estimation forms an important kernel routine in our memory exploration environment ([2]), which is an analytical platform that helps perform an application specific memory customization for embedded processor-based systems.

## References

[1] P. R. Panda, N. Dutt, and A. Nicolau "Data Cache Sizing for Embedded Processor Applications," TR-ICS-97-31, U. C. Irvine, 1997.

[2] P. R. Panda, N. Dutt, and A. Nicolau "Architectural Exploration and Optimization of Local Memory in Embedded Systems", 10th International Symposium on System Synthesis, September 1997.

[3] P. R. Panda, N. Dutt, and A. Nicolau "Memory Data Organization for Improved Cache Performance in Embedded Processor Applications," ACM TODAES, Vol. 2, No. 4, October 1997.