Multiprocessor System-on-Chip Data Reuse Analysis for Exploring Customized Memory Hierarchies[*]

Ilya Issenin
University of California,
Irvine, CA 92697
isse@ics.uci.edu

Erik Brockmeyer
IMEC, B-3001
Leuven, Belgium
brockmey@imec.be

Bart Durinck
IMEC, B-3001
Leuven, Belgium
bart.durinck@imec.be

Nikil Dutt
University of California,
Irvine, CA 92697
dutt@ics.uci.edu

ABSTRACT

The increasing use of Multiprocessor Systems-on-Chip (MPSoCs) for high performance demands of embedded applications results in high power dissipation. The memory subsystem is a large and critical contributor to both energy and performance, requiring system designers to perform exploration of low power memory organizations. In this paper we present a novel multiprocessor data reuse analysis technique that allows the system designer to explore a wide range of customized memory hierarchy organizations with different size and energy profiles. Our technique enables the system designer to explore feasible memory subsystem solutions that meet power and area constraints while maintaining the necessary performance level. Our experiments on the complex QSDPCM benchmark illustrate the exploration of a wide range of customized memory hierarchies for an MPSoC implementation.

Categories and Subject Descriptors

C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems - r*eal-time and embedded systems.*

General Terms:
Algorithms, Performance, Design.

Keywords:
Scratch pad memory management, customized memory hierarchy, multiprocessor data reuse analysis.

1. INTRODUCTION

Image and video processing applications typically have a lot of parallelism and thus are perfect candidates for being parallelized and executed on Multiprocessor Systems-on-Chip (MPSoC). In addition to easy parallelization, such applications typically have regular memory access patterns that can be statically analyzed and predicted at compile time. This property allows us to replace caches that are used in general-purpose systems by scratch pad memories (SPMs). SPMs are more power efficient than caches because of the absence of tag memory, tag comparators and hardware that enforce cache coherency. Furthermore, SPMs allow easier and tighter estimation of WCET bounds of real-time behaviors.

However, in order to use SPMs, compile-time analysis of the application is required so that the necessary transfers between memories are effected and coherency is maintained if the same SPM is used by several processors.

To the best of our knowledge, our work is the first to propose a multiprocessor data reuse analysis technique that is able to generate a number of scratch pad memory hierarchies for a given multiprocessor code, thus enabling the exploration of the power-area space of customized memory hierarchies. Increasing the number of SPM-based memory subsystem design alternatives available to designer allows the use of a trade-off solution that meets the design constraints which are not feasible otherwise.

2. RELATED WORK

Related work has appeared in several categories.

In uniprocessor systems, many of the approaches use scratch pad memories to store the most frequently used *scalars* or the *whole arrays* [6] [9] [11] [12], where placement of part of array in the scratch pad memory is not possible.

However, our technique is geared towards run-time placement of currently active *array elements* in scratch pad memories and thus is complementary to the works which are focused on placement of non-array data structures. It should be noted that typical multimedia applications employ large arrays and use computations that cannot fit all the desired data structures into local memory. Thus we believe it is critically important to develop strategies like ours that handle accesses to arrays that reside in large and expensive main memory. Several strategies were proposed to perform run-time prefetching for large streaming data, e.g. [4]. However, this approach only works with simple access patterns and do not allow to build memory hierarchies.

Among approaches targeting multiprocessor systems, Ozturk et al. [5] proposed to customize on-chip multiprocessors with private and shared single-level memories which are used similar to a cache. Chen et al. [1] proposed a strategy for mapping the data to local single-level SPMs in a multiprocessor system.

In contrast, our approach allows to generate a number of possible multi-level SPM hierarchies consisting of shared and private scratch pads with different overall area/power/performance trade-offs, giving designers additional freedom in customizing the memory subsystem.

3. MEMORY MODEL

In our approach we consider a multiprocessor system consisting of several processors and shared or private scratch pad memories. Our technique handles all data accesses with regular access pattern (i.e., vector accesses), which constitute a large portion of accesses in typical embedded multimedia applications. We assume that there is a mechanism to handle all other data memory accesses. For example, all local variables and stack can be placed in local memories of each of the processor. If there is a need to use bigger (or shared) non-local memory, caches supporting coherency

[*] This work was partially supported by NSF grants CCR-0203813 and CCR-0205712.

protocol could be used by each processor to handle such irregular requests.

4. SYNCHRONIZATION MODEL

When several processors use the same data, it may be beneficial to implement a buffer in a memory that can be accessed by both processors instead of duplicating the data in local memories. We call such a buffer a *shared buffer*.

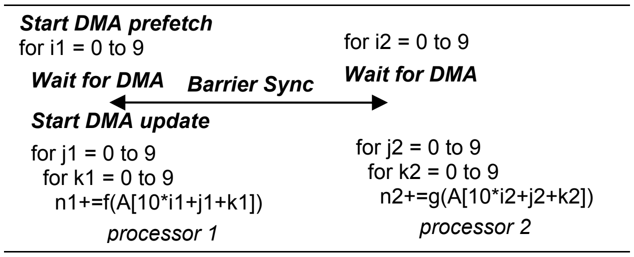
```
for i1 = 0 to 9                          for i2 = 0 to 9
  for j1 = 0 to 9                          for j2 = 0 to 9
    for k1 = 0 to 9                          for k2 = 0 to 9
      n1+=f(A[10*i1+j1+k1])                    n2+=g(A[10*i2+j2+k2])
            processor 1                              processor 2
```

Figure 1. Code fragment using same data

Figure 1 shows an example of code where two processors are using the same array elements. While it is possible to duplicate array A and store it in local memories, this would require 200 accesses to the main memory. However, if we introduce a small shared scratch pad memory, it is possible to reduce the number of accesses to main memory to 100.

In this configuration the shared buffer is updated on every iteration of the outermost loop (*i*). From this example it should be clear that the presence of a shared buffer does not allow processors to run independently, even if there are no data dependencies between processors in the original program. Now both processors have to synchronize their execution before updating the shared buffer, otherwise the buffer update initiated by one of the processors may expel the data from a buffer that has not been read yet by the second processor.

One way to implement such synchronization is by using barrier synchronization and increase the size of the buffer so that the DMA could prefetch data to one part of the buffer while the processors are using data from the other part (Figure 2).



```
Start DMA prefetch
for i1 = 0 to 9                          for i2 = 0 to 9
  Wait for DMA     Barrier Sync          Wait for DMA
  Start DMA update
  for j1 = 0 to 9                          for j2 = 0 to 9
    for k1 = 0 to 9                          for k2 = 0 to 9
      n1+=f(A[10*i1+j1+k1])                    n2+=g(A[10*i2+j2+k2])
          processor 1                              processor 2
```

Figure 2. Example of buffer updates and processor synchronizations implementation

In the rest of the paper we present our approach for detecting data reuse in a multiprocessor system and show how it can be applied for building highly customizable scratch pad based memory subsystems.

5. DRDM: DATA REUSE DETECTION FOR MULTIPROCESSORS

Algorithm 1 in Figure 3 outlines our Data Reuse Detection for Multiprocessors (DRDM) approach. First, we obtain a code parallelized for execution on several processors and determine time-critical and memory intensive parts (kernels) that should determine the final memory configuration. Since we perform compile-time analysis and an arbitrary program cannot be statically analyzed, we assume that the memory accesses have regular access pattern that can be expressed as affine functions of the outer loop iteration (a reasonable assumption for multimedia applications). Figure 1 shows an example of such code. Our technique takes a description of loops and array index functions for each of the processors as the input. If the code does not contain explicit index functions but has a behavior that can be described with them, an approach such as [3] can be used to convert the code into this form.

Algorithm 1: Data Reuse Detection for Multiprocessors (DRDM)

```
1. Map the application to processors and select parts of the
   program to be optimized; extract loop and array reference
   information
2. Select synchronization granularity. For each selection do:
   2.1. Convert multiprocessor reuse analysis problem into
        uniprocessor reuse analysis problem
   2.2. Apply uniprocessor data reuse analysis technique to
        the obtained uniprocessor problem
3. Combine obtained reuse trees into one multiprocessor
   reuse graph
```

Figure 3. Our approach for program analysis

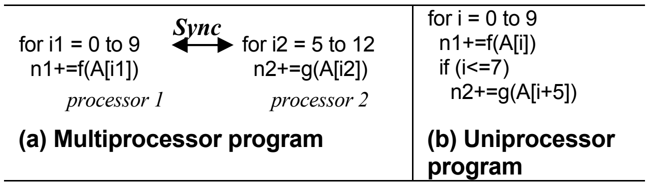
In Step 2 of Algorithm 1 we select at which loop level the processors should be synchronized (an example of synchronization was discussed in Section 4; in that example the processors were synchronized at loop *i*). This granularity selection trades off the overhead caused by the additional synchronizations by the buffer size (typically synchronization at the inner loops leads to a higher number of synchronizations and smaller buffer size). Since a trade-off is present, it is beneficial to keep alive more than one possible synchronization decision and do further analysis (Steps 2.1 and 2.2) on each of them, including the case when no synchronizations are present. Note that in case of more complex loop structures (other than perfect loop nests) it is possible to have more than one synchronized loop at the same time; these loops, however, should not be one inside the other to exclude the possibility of deadlocking.

There may be many possible ways to synchronize the loops. However, not all of them are equally useful and thus less useful ones should be pruned. For example, for the program shown in Figure 1, the pair *j1-i2* (i.e., during the first iteration of loop *i1* on processor 1, each iteration of loop body *j1* is synchronized with the iteration of the loop body *i2* on the processor 2) is not useful, since it leads to poor load balancing. We found that on typical applications only few options are left after applying a set of computationally simple pruning rules.

In Step 2.1 of Algorithm 1 we convert the multiprocessor data reuse analysis problem into uniprocessor analysis problem using the following observation as illustrated by Figure 4.

The program shown in Figure 4a has two loops executed on two processors which are synchronized as explained in Section 4. During the first iteration of the loop elements A[0] and A[5] are accessed simultaneously (if the shared memory with the array A has only one port, the elements are accessed in arbitrary order but during the first loop iteration). On the second iteration elements A[1] and A[6] are accessed, and so on. The functionally equivalent uniprocessor program in Figure 4b accesses the same elements but in fixed order (first A[0], then A[5] during the first iteration, and so on). However, from the reuse analysis point of view, both programs have identical data reuse patterns. This suggests that we can apply uniprocessor data reuse analysis to multiprocessor programs by merging programs. During such a merge all the loops above (and including) synchronized loops are replaced by the set of loops from the first processor and the inner loop bodies are added sequentially.

The constant term in the index expressions may need to be adjusted, as well as the upper loop bounds (as was done in Figure 4). Note that the *if* condition that was added to the uniprocessor program in Figure 4b to preserve functionality is not needed since it doesn't change the reuse pattern.


```
for i1 = 0 to 9    <-Sync->   for i2 = 5 to 12
  n1+=f(A[i1])                  n2+=g(A[i2])
      processor 1                   processor 2
```
(a) Multiprocessor program

```
for i = 0 to 9
  n1+=f(A[i])
  if (i<=7)
    n2+=g(A[i+5])
```
(b) Uniprocessor program

Figure 4. Multiprocessor and uniprocessor programs with the same data reuse pattern

After obtaining the uniprocessor code, it can be analyzed in Step 2.2 of Algorithm 1 using any technique for uniprocessor data reuse analysis; we apply the technique described in [2]. The result of the analysis is a hierarchical set of buffers, also called a *reuse tree*. Each buffer in reuse tree can be mapped to a physical SPM or left unimplemented.

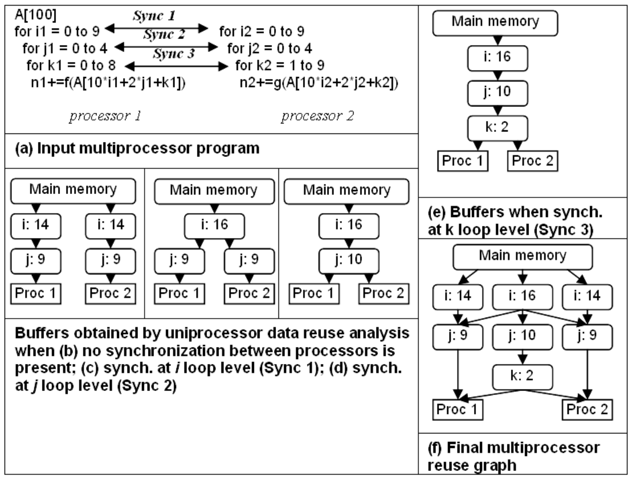
In Step 3 of Algorithm 1 we combine the buffers obtained by uniprocessor data reuse analysis for different synchronization selections into one graph. The Figure 5 shows an example of multiprocessor program with possible synchronization choices (Figure 5a), reuse trees obtained by uniprocessor reuse analysis for these different synchronization selections (Figure 5b-e) and the combined final reuse graph (Figure 5f). The numbers inside the boxes representing buffers are the sizes of the buffers suggested by uniprocessor reuse analysis. Note that not all buffers in the reuse graph can be implemented simultaneously in scratch pad memories. For example, on the *i* loop level, a shared buffer (of size 16) and the processor 1's private buffer (of size 14) are exclusive alternatives and are not meant be implemented simultaneously. The general rule is that if two buffers in the reuse graph are on the same loop level and have common descendants, they are mutually exclusive.

Finally, after applying our DRDM technique described above, the designer is provided with many more options for implementing memory subsystem than just having one memory for holding the array.

The advantage of having intermediate buffers is not only in exploiting data reuse, i.e., in reducing the number of main memory accesses (and thus saving time and energy). Typically single accesses to fetch data are much more expensive than block (DMA) transfer of large amount of data at once. Our approach allows implementation of an efficient prefetch scheme, where data is brought as close to the processor as possible by block transfers, and the processors make single accesses only to the closest (local) memory. This scheme is efficient since the memory latency can be completely hidden (assuming there is enough bandwidth available) and for the processor all the fetches of the data from the main memory will have the latency of small local memory.

6. EXPERIMENTAL RESULTS

In this section we perform a study of a multiprocessor implementation of the QSDPCM video encoder [10] and show that our technique enables both a finer granularity of memory subsystem customization as well as a larger space of alternatives explored than would be possible without our technique.


```
A[100]
for i1 = 0 to 9   <-Sync 1->   for i2 = 0 to 9
  for j1 = 0 to 4  <-Sync 2->    for j2 = 0 to 4
    for k1 = 0 to 8 <-Sync 3->     for k2 = 1 to 9
      n1+=f(A[10*i1+2*j1+k1])        n2+=g(A[10*i2+2*j2+k2])

      processor 1                    processor 2
```
(a) Input multiprocessor program

Buffers obtained by uniprocessor data reuse analysis when (b) no synchronization between processors is present; (c) synch. at *i* loop level (Sync 1); (d) synch. at *j* loop level (Sync 2)

(e) Buffers when synch. at k loop level (Sync 3)

(f) Final multiprocessor reuse graph

Figure 5. Combining the buffers obtained by uniprocessor data reuse analysis

In our experiments, we used the internal IMEC multiprocessor simulation environment that simulates application execution on multiple TI C62 DSP processors. We used it to obtain the number of accesses to private L1 4K processor memories. The number of accesses to other memories was calculated using benchmark loop structure information and buffer parameters. We used the CACTI model [7] for obtaining L2 memory access energy for 130 nm technology. In our memory energy model we did not account for bus energy dissipation nor for the difference in energy consumption of private and shared memories. For the 64MB off-chip L3 (main) memory, we used the energy and bus model published in [8], with the total energy of 42 nJ per 32-byte transfer.

The QSDPCM encoder applicsation consists of several parts: subsampling and motion estimation with 4, 2 and 1 pixel accuracies (ME4, ME2, ME1) and quadtree decomposition and reconstruction (QC). The application was mapped into 6 processors as shown in Figure 6, guided by load balancing considerations. The processing of the frame is pipelined with the total of three pipeline stages (Figure 6). On each stage the computation is performed on one line of 16x16 pixel blocks. In our experiments we used frames with VGA resolution (40x30 blocks).

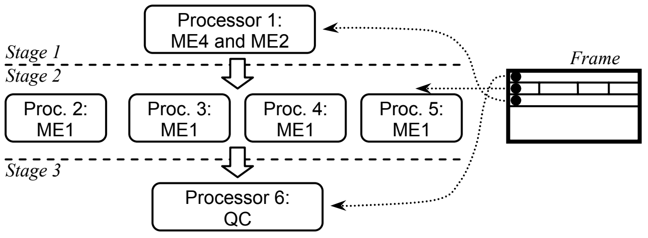


Figure 6. The mapping of the application to the processors

After analyzing the application we identified several arrays that are large enough (or that are shared among the processors) so that they cannot be placed in the local 4K processor memories. We performed analysis of the accesses to these arrays using our technique described in Section 5. Reuse graphs obtained by our approach are shown in Figure 7. Recall that each reuse graph represents a hierarchy of buffers; each buffer can be mapped to one of the SPMs or not implemented at all. A set of buffers mapped to SPMs represents a potential customized SPM hierarchy with its own performance, area and energy costs. The graphs were pruned to exclude the buffers with the sizes less than 40 bytes since it is not practical to implement them.

To completely hide access latencies to all data that are not residing in memory layer 1 of each of the processor, it is necessary that all the data is prefetched to L1 memory ahead of time. That is why we fixed the mapping of all smallest buffers in the reuse graphs to L1 (as we mentioned earlier, all local variables and stack are also mapped to L1). In this case the processor never accesses any other memories except L1, and all transfers to L1 are performed by the DMA controllers.

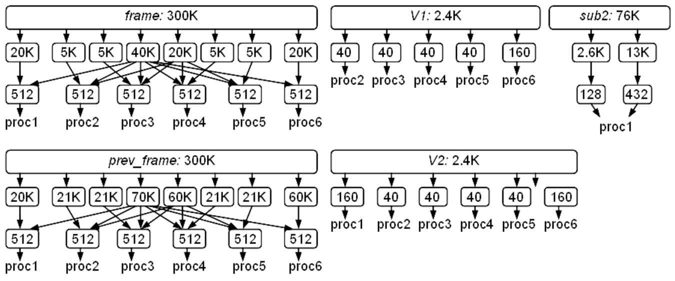


Figure 7. Reuse graphs obtained by our approach

Even in this case, when the mapping of smallest buffers is fixed, there are a lot of choices in mapping of the rest of the buffers to private or shared L2 or main memories. For each of the reuse graphs in Figure 7 we selected several different mappings, including mapping of the whole arrays to the L2 SPM for the arrays smaller than 256K. We calculated the energy consumed by the array accesses for each mapping, and selected Pareto optimal points (energy vs. sum of buffer sizes). Then, for each combination of these Pareto optimal mappings, we calculated the total application memory subsystem energy consumption.

The results are depicted in Figure 8. Pareto points are circled on the graph. We can see that the mapping decisions have a significant impact on the area and energy consumption of the design. Among the Pareto optimal points of **the graph the L2 buffer size varies from 0 to 190K and the total memory subsystem consumption changes within 33% depending on the SPM configuration**, clearly demonstrating the utility of our reuse analysis technique for enabling MPSoC memory hierarchy exploration.

Without the use of our technique, it would be only possible to have design solutions marked by rectangles in Figure 8. It is important to note that 1) not all the design alternatives obtained without using our approach (mapping the whole arrays to SPMs) and marked by rectangles are Pareto optimal, and 2) our data reuse technique opens up a much wider range and finer granularity of size and energy trade-offs would not have been explored otherwise. Without the use of our technique, the size varies from 0 to 80K and the total memory subsystem consumption changes only within 7%.

7. CONCLUSION

The increasing use of MPSoCs places a big burden on system designers to evaluate customized memory hierarchies that yield different power, cost and performance trade-offs. In this paper we presented a novel multiprocessor data reuse analysis technique that effectively allows designers to explore customized memory hierarchies in MPSoCs, and which opens up a large space of heretofore unexplored options for memory customization.

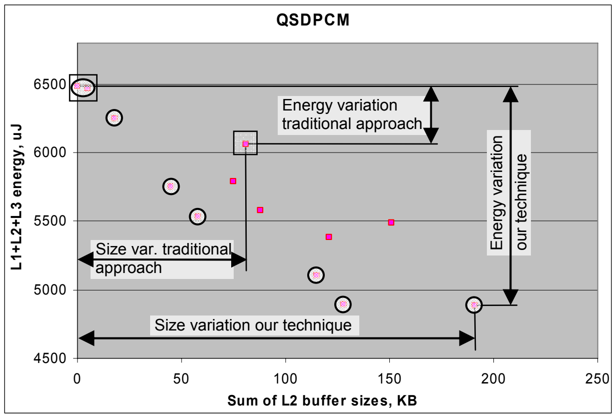


Figure 8. Design alternatives provided by out technique

Our experiments on the complex QSDPCM benchmark clearly demonstrate the utility of our approach. The use of our technique increases the design space by two times on the size dimension and increases the energy range by three times in comparison with traditional approaches.

Future work will investigate heuristics to explore all the alternatives provided by our technique.

8. REFERENCES

[1] G. Chen et al. Exploiting Inter-Processor Data Sharing for Improving Behavior of Multi-Processor SoCs. ISVLSI 2005.

[2] I. Issenin et al. Data Reuse Analysis Technique for Software-Controlled Memory Hierarchies. DATE 2004.

[3] I. Issenin, N. Dutt. FORAY-GEN: Automatic Generation of Affine Functions for Memory Optimizations. DATE 2005.

[4] J. Lee et al. Memory Access Pattern Analysis and Stream Cache Design for Multimedia Applications. ASP-DAC 2003.

[5] O. Ozturk et al. Customized On-Chip Memories for Embedded Chip Multiprocessors. In Proc. ASP-DAC 2005.

[6] P. Panda et al. Efficient Utilization of Scratch-Pad Memory in Embedded Processor Applications. In Proc. DATE 1997.

[7] P. Shivakumar, N. Jouppi. CACTI 3.0: An Integrated Cache Timing, Power, and Area Model. WRL Technical Report 2001/2, Aug. 2001.

[8] A. Shrivastava et al. Compilation techniques for energy reduction in horizontally partitioned cache architectures. In Proc. CASES 2005, San Francisco, California.

[9] S. Steinke et al. Assigning Program and Data Objects to Scratchpad for Energy Reduction. In Proc. DATE 2002.

[10] P. Stobach. A new technique in scene adaptive coding. In Proc. EUSIPCO, Grenoble, 1988.

[11] S. Udayakumaran, R. Barua, Compiler-Decided Dynamic Memory Allocation for Scratch-Pad Based Embedded Systems. In Proc. CASES, San Jose, CA, 2003.

[12] M. Verma et al. Dynamic Overlay of Scratchpad Memory for Energy Minimization. In Proc. CODES 2004.