SAME 2000

Session: 1 Codesign

Memory Optimization of Data Flow Applications at the codesign level

Antoine Fraboulet, INSA Lyon, France
Laurence Just-Meunier, Cadence Design Systems, Sophia-Antipolis, France
Anne Mignotte, INSA Lyon, France

Abstract:

Portable or embedded systems as well as submicronic technologies have made the power consumption criteria crucial. Memory is known to be extremely power consuming. Moreover multimedia applications are memory intensive applications. Therefore we propose new techniques to optimize a behavioral description of multimedia applications before the Hardware/Software partitioning (Codesign). These transformations are performed on "for" loops that constitute the main parts that handle the arrays of the multimedia code. These optimizations are then tested within the VCC (Cadence ® Virtual Component CoDesign) framework.

1 Introduction

Code transformations for the design of an integrated system can be performed at several levels. For instance, Boolean functions minimization can be considered. At the Register Transfer Level, transformations are performed on the control state charts by splitting or merging nodes. At every stage of the design we can apply transformations used in software compilers [1].

In this paper, we will focus on the design of data flow embedded systems. These systems use signals and data stored in arrays such as images, video and sounds. This type of applications consumes memory for multidimensional data storage. More than a half of the surface of integrated systems of this kind is filled by memory. Memory is known to be power consuming. Therefore, this massive memory made power consumption criteria control compulsory. Power optimization by memory optimization can be done in several ways: the reduction of the size of the memory and improved data-movement strategies over the memory hierarchy.

Once the Hardware-Software partitioning is done, the memory has already been divided. It is therefore very important to make optimizations before this partitioning in order to deal with all the memory in homogeneous vision. In this paper, we want to optimize both types of memories, the one that will be included in hardware and the one controlled by software.

Methods and tools defined in this paper are specific to Codesign because in the case of software-only compilation the memory is already instantiated in hardware and cannot be tuned. Moreover, the memory is rarely shared in software; a memory cell is used for only one array cell (unless explicitly stated when the designer uses the same name for two non-overlapping objects).

On the opposite, during silicon compilation the physical memory is optimized all along the design chain. In-Place Mapping [2] can allow to share memory by overlapping arrays when possible, memory allocation can select memory modules upon several criteria (size, number of ports) and assignment computes hardware addresses for accessing arrays in memory [3]. The memory is instantiated on-demand and specific modules can be used or even built specifically. This will be the main assumption for memory optimization by code transformations.

All the optimizations undertaken during silicon compilation improve the design starting from a given description. However, a preprocessing source-to-source code transformation similar to the one used in software compilation can be applied on the design in order to improve the efficiency, or enable, further optimizations. The source-to-source transformations we propose are target independent code optimizations. We do not consider specific modeling of the target except that it has, at least, two levels of memory. These transformations are good on general principles and are a complement to transformations that are designed for a specific target platform.

The handling of arrays is done mainly through "for" loops in multimedia applications. These loops form the critical part of the optimizations we want to apply at the Codesign stage. We propose to transform the algorithmic description of a design by exploiting and adapting techniques similar to the ones used in automatic parallelization [4,5,6] (see also [7] and [8]) so as to reduce the consumption in power due to memory by enabling powerful optimizations.

In this paper, we will present "for" loop transformations to optimize memory. The following section defines the basic target architecture and the memory criteria to reduce power consumption. Section 3 introduces "for" loop transformations and develops two main transformations, namely loop merging and loop alignment. There was a lack of environment and experiments results that prove the effectiveness of the proposed approach.
Our goal with the section 4 is to demonstrate that the optimization performed at the codesign level can be tested in the VCC framework.
VCC methodology is leveraging the selection of the most appropriate optimization by ranking several alternatives in term of performance and cost. The goal is the estimation memory traffic in the architecture. We'll focus on the estimation of components' activation, such as CPU, Cache and Memory, number of memory's access over time or statistics, bus and memory's load.

2 Memory optimizations in Multimedia Embedded systems

Memory operations consume far more power than control or arithmetical operations. Experiments have shown the relative cost of a memory operation compared to arithmetic computations; for example a transfer from an external memory consumes 33 times more than a 16-bit addition. Experimentation made at IMEC [9] on MPEG4 has shown important consumption gains by code transformations on the algorithmic description of the design in *C* language (decrease of a factor 4 on average consumption and of a factor 10 on peak power).

2.1 Architecture and Memory Hierarchy

Target multimedia architectures and applications impose a complex memory hierarchy: registers, hardware and/or software caches, on-chip or off-chip memory [10]. We use a simplified view of the target architecture for codesign shown on Figure 1. The only feature we consider here is the availability of a foreground (on-chip) and a background memory (off-chip) with a communication bus between them.

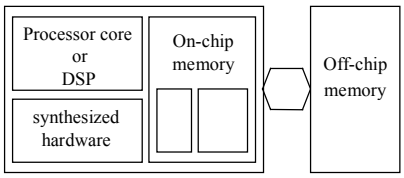


Figure 1: Target architecture

2.2 Cost Functions and Models

The size of the memory needed by an application is approximated by the maximum size of time overlapping arrays. This is based on the assumption that the memory will be shared afterward by In-Place Mapping [2]. It gives the following definition:

$$MemSize = \max_{\forall t}\left(\sum_{A \in Live(t)} Size(A) \right)$$

Size (A) is the size of the array *A* and *Live (t)* is the set of arrays that contains useful data at time *t*.

The power consumption of a memory access increases with the level from which the data has to be fetched. An access to an external memory consumes more power than an access to an on-chip memory (for instance, some experimentation show that an access to internal memory of a system consumes 3 times less than an access to the external one). Memory hierarchies are well exploited if we can achieve a good data *temporal locality*. This locality represents the amount of time between two successive accesses (either write-read or read-read). The average cost can be defined as follow:

$$\sum_{(i',i) \in DoubleAccess} |i' - i|$$

DoubleAccess is the set of index couples that corresponds to two successive accesses to the same array location.

We can refine these criteria depending on the context of the design. More precise hypotheses are needed on the memory hierarchy and memory management stages during the design to perform platform specific transformations. Memory hierarchy properties are fixed all along the design process. The criteria we use can benefit from already known features of the final system such as the depth of the hierarchy or the size ratio between consecutive level [13]. However the purpose here is to use generic transformations without specific knowledge of the target architecture.

3 Loops Transformations

"For" loops transformations change the way arrays are traversed all over the program. They allow removing some temporary arrays or variables and they can modify the memory access locality. In our

context, we focus on array optimizations. Only data dependencies are considered. Dependence analysis tools already exists for *Fortran* but recent research tools can also deal with a subset of *C* [12] (the main problem is the handling of pointers in *C*).

Methods are divided into two classes: global methods which deal with each loop as atomic computation unit and local methods which change the way loops are organized internally. We list here some global transformations that are useful for memory optimization:

- *Code moving* that changes the execution order between two loops in the program without modifying the loops.
- *Loop merging* that groups several loops in a unique one.
- *Loop splitting* that realizes the reverse of merging.

Local transformations explore more in depth the way loops are organized internally they can modify memory locality and space requirement.

- *Loop tiling or partitioning* increases the nesting level of a loop. The effect is that the iteration space is cut into blocks that are processed sequentially.
- *Loop Unrolling* decrease the number of iterations by describing several times the same instruction in the loop body.
- *Loop Pipelining* or *Alignment* or *Folding* shifts some instructions from one to several iterations within the loop body.
- *Loop collapsing or coalescing* is the reverse of tiling.

More details and examples on these loop transformations can be found in [6].

Silicon compilation experiences have shown that we cannot solve all the problems at a time without loosing the effect on each optimization criteria. In order to get a feeling on the transformations, we have chosen to develop two transformations. A first one that transforms the code globally: loop merging and its associated code moving techniques and a second one that is more local: loop alignment.

3.1 Loop Merging

Memory minimization by loop merging is obtained by reducing the size of temporary arrays. An array written in a loop and read in another one must be stored in memory between these operations. It is necessary to merge these two loops to free the used memory. An array can be removed if its last use in the program is in the same loop as its production.

```
L1:for i=1 to n
   a[i]=fun0(a[i-1],a[i-2])
   b[i]=fun1(a[i])
L2:for i=1 to n
   c[i]=fun2(b[i-1])
L3:for i=1 to n
   d[i]=fun3(c[i+1])
```

Figure 2: Initial Example Program

For instance array *b* in figure 3 can be replaced by a scalar in the loop body it is not used elsewhere in the program. Figure 3 represents the program of Figure 2 after merging the loops *L1* and *L2*.

```
L1L2: for i=1 to n
   a[i]=fun0(a[i-1],a[i-2])
   b[i]=fun1(a[i])
   c[i]=fun2(b[i-1])
L3: for i=1 to n
   d[i]=fun3(c[i+1])
```

Figure 3: Loop merging between *L1* and *L2*

The third loop *L3* cannot be merged with the others. Every edge that carries dependence can be removed by shifted loop merging. As we can see on Figure 3, loops *L1L2* and *L3* cannot be simply merged. We have to shift the third loop (producing *d[i-1]* instead of *d[i]* in order to merge *L1L2* and *L3*.

```
Prologue
L1L2L3: for i=2 to n-1
   a[i]=fun0(a[i-1],a[i-2])
   b[i]=fun1(a[i])
   c[i]=fun2(b[i-1])
   d[i-1]=fun3(c[i])
Epilogue
```

Figure 4: Shifted loop merging of *L1L2* and *L3*

This fusion produces a prologue and an epilogue. Loop boundaries have been reduced to fit the common iteration space of the merged loops. Another solution (not shown in this paper) would have been to introduce predicates in the loop body and extending the loop bounds to *[1,n+1]*. Experiments have shown that the overhead introduced by adding extra control or arithmetic statements (such as modulo expressions for indexing arrays) is negligible in front of the gain obtained over the memory. The proposed transformation is thus always acceptable if it can be used to free temporary arrays.

3.2 Loop alignment

Loop pipelining is used in automatic parallelization to reduce the latency of a loop when run on superscalar processors. This method allows decreasing the computation time of a loop. However, a similar transformation may be used to optimize the use of foreground memory.

```
Prologue
For i=3 to n
   a[i]=fun0(a[i-1],a[i-2])
   b[i-1]=fun1(a[i-1])
   c[i]=fun2(b[i-1])
   d[i-1]=fun3(c[i])
Epilogue
```

Figure 5: Alignment on the loop on Figure 4.

The algorithm we present in this section minimizes the size of the foreground memory needed to store values that are computed and used in the same loop. This algorithm is fully described in [14]. This minimization can also be seen as optimizing the average distance in terms of temporal locality between read and write accesses to the same variable in different loop iterations. This technique is useful to keep values in a memory near the top of the hierarchy (where memories are smaller and less power consuming) but it can also decrease the memory size needed by the application (a dimension of an array can be reduced to a scalar value for instance).

The alignment method can fine-tune a loop nest for memory size optimization. However if we have to deal with a lot of arrays in the same loop there are chances that we will run into trouble with memory caching strategies. Once all the loops are merged the resulting program might end up with a big loop nest that handles all the computation and all the arrays at the same time. An optimization method has to be defined to take care of the memory hierarchy. Accessing to 4 arrays at the same time, as in Figure 4, might end up with a lot a cache misses during the execution. To improve usage of all levels of the memory hierarchy we have to decompose the computation of the loop into subsets.

Tiling transformations are a good candidate for this optimization. However, at this stage of the design we do not know the properties of the memory hierarchy and we must define new criteria on the measure of locality for this.

4 The Cadence ® VCC Framework and Methodology

The VCC framework is made of:

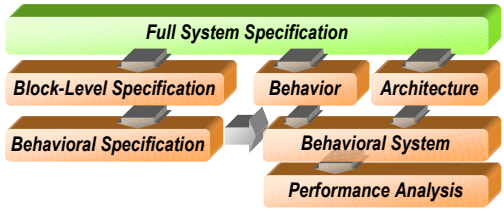
A behavior editor that is used to capture an unambiguous executable behavior specification at a high level of abstraction using behavior models from vendor libraries, exported models from other design environments, or user-defined behavior models.
An architecture diagram editor that captures a target architecture using high-level hardware and software architecture models and communication paths from vendor or user libraries.
A mapping editor that enables HW/SW and general architectural trade-off considerations by associating each model in the behavior diagram with a hardware or software architecture element and identifying its communication path as required.
A simulator that explores the design space by simulating the performance of each behavior as mapped onto an architectural element.
An analysis environment in which selected trace output can be charted and graphed to evaluate the performance of the mapped design.
Performance modeling techniques that provide delay mechanisms that are used to simulate the execution of each behavior on the mapped architecture.

The exploration methodology is divided into 3 major steps:

- Building a Virtual Prototype of the System Functionality
- Defining the Architectural Platform
- Mapping Function to Architecture

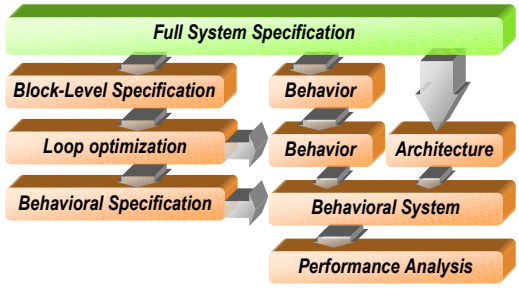
Then

- Analysis in a Graphical Environment



This framework supports the loop optimization experimentation. The methodology is thus applied with the following steps:

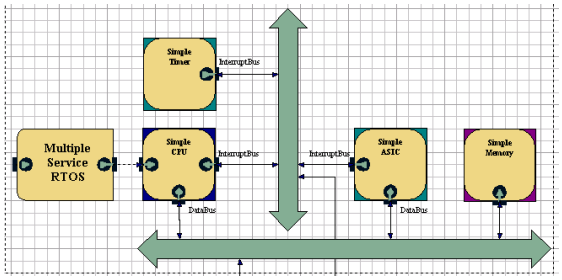
- Building a Virtual Prototype of the System Functionality
- Applying the loop optimizations
- Defining the Architectural Platform
- Mapping Function to Architecture



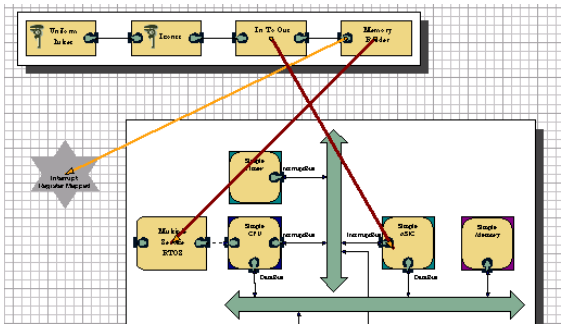
The VCC behavior diagram editor lets you capture the function of a system by creating a behavior diagram a collection of functional models that are

wired together. This system can be heterogeneous, In the loop optimization experimentation, the C and C++ flow will be explored.

Using the VCC architecture diagram editor designers capture the abstract target architecture onto which the system function will be mapped. Since it is a complete Co-design environment, the VCC environment supports essential architectural elements such as CPUs, DSPs, RTOSs, buses, memories, and dedicated HW and SW. To allow fast design evaluation, these architectural elements are modeled at a higher level of abstraction than implementation-level C or HDL. The chosen architecture is a computational resource (CPU) with his scheduling policy, a memory, a cache and buses (data and interruption).



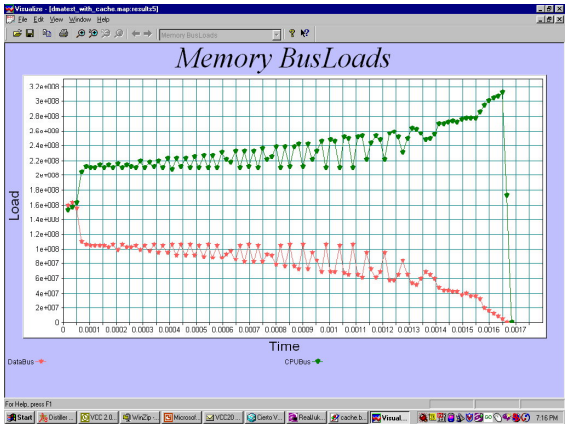
The VCC mapping diagram editor enables designers to map system functionality onto target architectural platforms between candidate hardware and software partitions and helps to identify the custom hardware needed to complete the system design.



Once a mapping diagram is completed, the system designers can evaluate the mapped design using performance simulation, which is enabled by software estimation and performance parameters that are annotated within timing free functional models.

The VCC environment offers comprehensive visualization techniques to display the results of functional and performance simulations. Users can easily customize two-dimensional, timeline, table and gantt charts to their particular analysis needs.

The probe in the VCC library will help to track the activation of the architectural elements such as buses and memories.



The memories within the VCC framework:

The memory and Cache are instantiated from the VCC architectural library. Then the transfers between architectural components are modeled with the architecture patterns and services. Architecture and pattern services are used to model communication that is consistent with the architecture elements and is consistent with the architecture topology.

A memory is a storage resource (e.g. ROM, RAM) or a memory mapped ASIC). A memory is divided into Memory segments.
The memory access is any activity referring to a memory. This can result from:

- an instruction/data read or write by a SW component
- a Post/Value on a port (in case of communication via shared-memory)

During simulation, VCC will find the path from the processor to the memory. Every memory access with is estimated through this path of busses (and bridges).
The command on the CPU for "Mapping Linker Segments" collects all the behaviors mapped to it (i.e. its RTOS) and then extracts the list of memory sections. This command then presents the uniquely named sections to the user and lets the user choose a memory participant and offset into that memory. The memory participant should be a memory in the architecture diagram.
VCC in most cases would automatically compute the size of the memory segment.
Memory hierarchy with different latencies, wait-states are modeled.

A cache is modeled by full simulation - Static analysis may not be acceptable

Every memory access is recorded
The access' address is [MemoryPortId, Offset]
The address is used to figure out the hit/miss:
If *hit*, the control is returned to CPU, the delay is the cache latency
If *miss*, the memory is accessed through the bus, the delay is cache+bus+memory latency.

The overall delay is not only the delay due to cache+bus+memory latency, but also the delay due to computational resources and schedulers.

5 Conclusions and future work

In this paper we are applying a codesign methodology to the loop optimization.
By associating an algorithm with an architecture, VCC is allowing the exploration of the whole system and the result analysis in term of activation of the architectural components, especially memories exploration, estimation of performance of the memory activation and bus loading.
Next step will be to actually apply this codesign methodology to an industrial application based platform.

References

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

[2] E. De Greef, "Storage Size Reduction for Multimedia Application", PhD Thesis, IMEC, 1998.

[3] M. Miranda, F. Catthoor, M. Janssen and H.De Man, "ADOPT: Efficient hardware address generation in distributed memory architectures", IEEE 9th International Symposium on System Synthesis, (ISSS'96), pp. 20-25, 1996.

[4] P. Feautrier, "Some Efficient Solutions to the Affine Scheduling Problem, Part I, One Dimensional Time", Int. J. of Parallel Programming, 21(5), 1992.

[5] P. Feautrier, "Fine-grain Scheduling under Resource Constraints", 7th Workshop on Language and Compiler for Parallel Computers, 1994.

[6] D.F. Bacon, S.L. Graham and O.J. Sharp, "Compiler Transformations for High-Performance Computing", ACM Computing Surveys, 26(4):345-420, 1994.

[7] PriSM SCPDP Team, "Systematic Construction of Parallel and Distributed Programs", World Wide Web document, http://www.prism.uvsq.fr/english/parallel/paf/autom_us.html.

[8] Stanford Compiler Group, SUIF Compiler System, World Wide Web document, http://suif.stanford.edu/suif/suif.html.

[9] E. Brockmeyer, F. Catthoor, J. Bormans, H. De Man, "Code transformations for reduced data transfer and storage in low power realisation of MPEG-4 full-pel motion estimation", Proceedings of the IEEE International Conference on Image Processing, ICIP Vol.3, pp. 985-989, Chicago, IL, October 1998.

[10] S. Chakravarty and G. Martin, "A new embedded system design flow based on IP integration", DATE 99 User's Forum, Munich, Germany, March 9-12, pp. 99-106, 1999.

[11] P.R. Panda, N. Dutt and A. Nicolau, "Memory Issues in Embedded Systems-On-Chip", Kluwer Academic Publishers, 1999.

[12] IMEC Acropolis Project, World Wide Web document, http://www.imec.be/vsdm/projects/mm_comp/.

[13] B. Alpern, L. Carter, E. Feig and T. Selker, "The Uniform Memory Hierarchy Model of Computation", Algorithmica, 12:72-109, 1994.

[14] A. Fraboulet, G. Huard and A. Mignotte, "Loop Alignment for Memory Accesses Optimization", IEEE 12th International Symposium on System Synthesis, (ISSS'99), pages 71-77, 1999.

[15] G. Martin and B. Salefski, "Methodology and Technology for Design of Communications and Multimedia Products via System-Level IP Integration", Proceedings of DATE 98 Designer Track, pp. 11-18, 1998.