Low power storage cycle budget distribution tool support for hierarchical graphs◇

Erik Brockmeyer, Arnout Vandecappelle, Sven Wuytack, Francky Catthoor*
Desics, IMEC, Kapeldreef 75, Leuven, Belgium

* Also professor at the Katholieke Univ. Leuven, Belgium
◇ Partly sponsored by the Esprit ESD-LPD project 25518 : DAB-LP

Abstract

In data dominated applications, like multi-media and telecom applications, data storage and transfers are the most important factors in terms of energy consumption, area and system performance. Several steps which optimize these costs are present in our systematic Data Transfer and Storage Exploration methodology. In the important step discussed in this paper, the cycle budget available for background storage transfers is globally distributed over the application's memory accesses that are typically grouped in the loop and function hierarchy. This is crucial for meeting the real-time constraints with a customized memory organisation without counteracting the memory size and energy budget optimizations achieved by earlier steps in our script.

This paper proves the effectiveness of the prototype tool on driver applications of several application domains. It clearly shows the tradeoff between power, area and speed.

1 Introduction

In data transfer and storage intensive applications, the memory accesses are often the limiting factor to the execution speed, both in custom "hardware" and instruction-set processors ("software"). Data processing can easily be sped up through pipelining and other forms of parallelism. On the other hand, memory bandwidth can be improved too, especially in customized memory organisations, but it is much more expensive to realize this. Multi-port memories cause a large cost in area and power and high-speed memories are restricted to very small sizes. However, they may not be avoidable in stringent timing constraints. Because memory accesses are so important, it is even possible to make an initial system level performance evaluation based solely on the memory accesses to complex data types. Data processing is then temporarily ignored except for the fact that it introduces dependencies between memory accesses.

Data Transfer and Storage Exploration (DTSE) [6] is a systematic methodology to optimize data dominated applications for memory size, energy consumption and bus load reduction. Here the "back-end" of the methodology is considered, which has the highest priority for tool support, whereas the "front-end" steps in the script are mainly code transformations which can still be done manually (but in a systematic way [17]). The DSE step, discussed in this paper, involve the design of a (partly) custom memory organization. The final memory organization should be as cost-efficient as possible, but on the other hand its performance must (just) meet the real-time constraints.

In our previous research we have found that the design of a custom memory organisation can be split up in a two steps. The first step trades off the memory bandwidth with the system real-time constraint (Storage Cycle Budget Distribution step), followed by second step to fit the memory architecture to the application's needs (Memory Allocation and Assignment step), taking into account the constraints generated in the first step [22] (Figure 1).

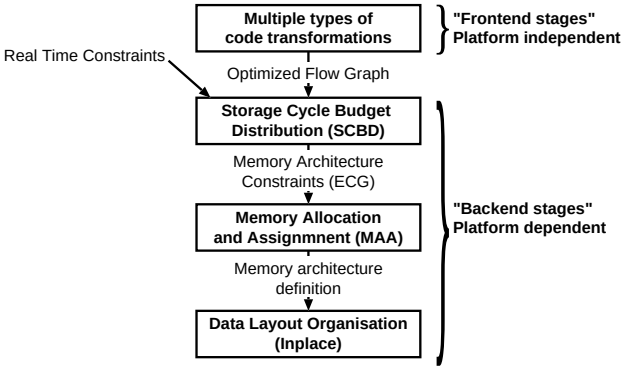

Figure 1. The memory architecture is defined in two steps (SCBD+MAA).

The approach taken in our Storage Cycle Budget Distribution (SCBD) method is to partly order the memory accesses such that the maximally required memory bandwidth is minimized [22]. This can be done by a (more) efficient use of the memory ports. In our application domain, an overall target storage cycle budget is typically imposed, corresponding to the overall throughput. In addition, other real-time constraints can be present which restrict the ordering freedom. The result of this SCBD step is a set of constraints for the memory architecture.

To carefully evaluate the effect of the SCBD step, a detailed custom memory architecture has to be decided. This objective is tackled in the Memory Allocation and Assign-

ment (MAA) step, for which a systematic technique has been published in [5, 17]. The MAA is done in three sub-steps. First, the number and type of memories is chosen, during the memory allocation sub-step. Then, every signal is assigned to one of the allocated memories in the signal-to-memory assignment sub-step. This also determines the dimension of each memory. Finally, the related bus and port organization are derived. After these sub-steps, the memory architecture is fully determined. This allows to derive quite accurate data transfer and storage related costs in term of power, area and speed. The actual data organization inside each memory is still partly free however and is only determined in the last step of the DTSE script, memory data layout.

Our previously published SCBD prototype tool supports the definition of a memory organisation which is cheap in terms of power and area [22, 5]. An overview of this technique is given in Section 4. More related work can be found in Section 2. However, this methodology and tool are applicable for flat flow graphs only. All real-life applications contain loops and a function hierarchy though which could only be handled locally one at a time. To overcome this undesired situation we have now significantly extended the current approach and tool. Section 5 emphasizes the most crucial additional problems which arise for hierarchical flow graphs. In Section 6 we present our incremental approach which gives a solution to these problems. The experiments in Section 7 have been performed with the aid of a prototype tool implementation to support this approach.

2 Related work

In the register allocation domain, the allocation techniques start from a fully scheduled flow graph and are scalar oriented. A nice literature overview of this domain, which is fairly well understood by now, can be found in [16]. Many of these techniques construct a scalar conflict or compatibility graph and solve the problem using graph coloring or clique partitioning. This conflict graph is fully determined by the schedule which is fixed before. This means that no effort is spent to come up with an optimal conflict graph.

In the less explored domain of memory allocation and assignment for hardware systems, the current techniques start from a given schedule [11], or perform first a bandwidth estimation step [1] which is a kind of crude ordering that does not really optimize the conflict graph either. These techniques have to operate on groups of signals instead of on scalars to keep the complexity acceptable, e.g., the *stream model* of Phideo [11] or the *basic sets* in the ATOMIUM environment [1].

In the scheduling domain, the techniques for optimizing the number of resources given the cycle budget are of interest to us. Also here most techniques operate on the scalar-level, e.g. [13, 18]. The only exceptions currently are the Phideo stream scheduler [19] and the Notre-Dame rotation scheduler [14]. Many of these scalar techniques try to reduce the memory related cost by estimating the required number of registers for a given schedule. Only few of them try to reduce the required memory bandwidth, which they do by minimizing the *number* of simultaneous data accesses [18, 19]. They do not take into account *which* data is being accessed simultaneously.

The main difference between our SCBD and the related work discussed here is that we try to minimize the required memory bandwidth in advance by optimizing the access conflict graph for groups of scalars within a given cycle budget. We do this by putting ordering constraints on the flow graph, taking into account *which* data accesses are being put in parallel (i.e., will show up as a conflict in the access conflict graph).

3 Speed-cost tradeoff considerations

The SCBD and MAA tool combination explores different solutions in the performance, power and area space. Indeed, SCBD generates many valid solutions for different cycle budgets and MAA generates multiple solutions in the power/area space for this cycle budget. Therefore it becomes possible to make the right tradeoff within this solution space. Of course the tradeoff can be based solely on the tool output when the application is data dominant. If the application is not fully data dominated, memory accesses cycles can be traded to data-path cycles. Also cycles assigned to tasks, in a complex application consisting out of multiple task, can be supported by this type of information. These kind of considerations illustrated with examples can be found in [3, 4].

4 Summary of the flat flow graph technique

The main goal of the approach is to find which arrays must be stored in different memories in order to meet the cycle budget (for details see [5, 22]). Ordering the memory accesses within a certain number of memory cycles determines the required memory bandwidth. The concept of memory cycles does not have to equal the (data-path) clock nor the maximum access frequency of the memory, it is only used to describe the relative ordering of memory accesses. Every memory access is assumed to take up an integer number of abstract cycles. A memory access can only take place in a cycle after any access on which it depends. The found ordering does not necessarily have to match exactly with the final schedule (complete scheduling, including data-path related issues). It is produced to make sure that it is possible to meet the real time constraints with the derived memory architecture.

If two memory accesses are ordered in the same memory cycle, they are in conflict and parallelism is required to perform both accesses. These simultaneous memory accesses can be done in two different memories, or, if it is the same array in a dual port memory. Thus, the conflict constrains the signal to memory assignment and incurs a certain cost. A tradeoff has to be made between the performance gain of every conflict and the cost it incurs [3, 4]. On the one hand, every array in a separate memory is optimal for speed and seemingly also for power. But having many memories is very costly for area, interconnect and complexity and due to the routing overhead also for power in the end. Due the to presence of the real time constraints and the complex control and data dependencies, a difficult tradeoff has to be made. Therefore automatic tool support is crucial.

Our previous tool orders by "balancing the flow graph" a single (flat) flow graph within the given time constraints. The flow graph is extracted from a C input description. Internally, it constructively generates a (partial) memory access ordering steered by a sophisticated cost model which incorporated global tradeoffs and access conflicts over the entire algorithm code. The cost model is based on size, access frequency and other high level estimates (eg. possibilities for array size reduction [8]). The technique used for this is to order the memory accesses in a given cycle budget by iteratively reducing the intervals of every memory access (starting with ASAP and ALAP) [22]. The interval reductions are driven by the probability and cost of the potential conflicts between the accesses.

An Extended Conflict Graph (ECG) is generated which follows out of the memory access ordering. It contains the conflicting arrays which are accessed simultaneously. These arrays need to be stored in different memories. Note that many possible orderings (and also schedules) are compatible with a given ECG. A consolidation of the memory organisation is needed in the subsequent memory allocation and assignment step (as explained in Section 1).

5 Extension for hierarchical graphs

The technique mentioned above can only be directly used for flat flow graphs, i.e. when no loops and no function calls are present. Loops could be unrolled and function calls could be inlined, but this destroys the hardware or code reuse possibilities present in the original specification. Moreover, the complexity would explode. Therefore, ordering the memory accesses has to be done hierarchical.

We define a *block* as a code part which contains a flat flow graph. It can contain multiple basic blocks and condition scopes. Even the function hierarchy can be adapted to the needs of ordering freedom. In practice, the term loop body and block coincide; all loop bodies are defined as a separate block and the body of a nested loop is part of an other block.

Because the number of iterations to the blocks is mostly different, the problem is increased in two directions. First, the distribution of the global cycle count over the blocks need to be found (see Section 5.1), within the timing constraints and while optimizing a cost function. Second, a single memory architecture must satisfy the constraints of all blocks, and therefore the global ECG cost must be minimized. Combining all the conflicts of the locally optimized blocks in the global conflict graph will lead to a poor result (see Section 5.2). Reuse of the same conflict over different blocks is essential.

In our application domains, an overall throughput constraint (maximal or average) is put forward for the entire application. For instance, in a video application the timing constraint is 40ms to arrive at 25 frames per second. Sometimes, additional timing constraints are given. In this paper, we will only deal with one global timing constraint but extensions to support additional internal constraints are feasible on top of this.

5.1 Cycle distribution across blocks

The distribution of the cycles over the blocks is crucial. A wrong distribution will produce a too expensive memory architecture because the memory access ordering cannot be made nicely in some of the blocks while there are "cheap" cycles available in other blocks. Every single block affects the global cost of the memory subsystem. Therefore, if cycle budget of one block is too tight (while there is space in other blocks) it will cause additional cost.

The global cycle budget can be distributed over different blocks in many different ways, as shown Figure 2. At the left side of the figure a program is given containing 3 consecutive loops, to be ordered in 500 cycles. The table of Figure 2 shows three different distributions and a good ordering matching the distribution. The resulting conflict graph and cheapest memory architecture is given in the last two rows. Obviously, the second solution (*loop-i* 2 cycles, *loop-j* 1 cycle and *loop-k* 2 cycles) is the cheapest solution. The very poor third distribution even forces a dual-ported memory (due to assignment of one cycle for *loop-i*).

The illustration here is based on an academic example to show the problem. But in fact, the problem in real-life applications is much more difficult. First, because more signals, accesses and blocks are involved, the number of different possible distributions increases. Second, the number of block iterations will not be equal for each block. The impact on the global time elapse of one block is much bigger than another block. Hence, there is more freedom in the search space.

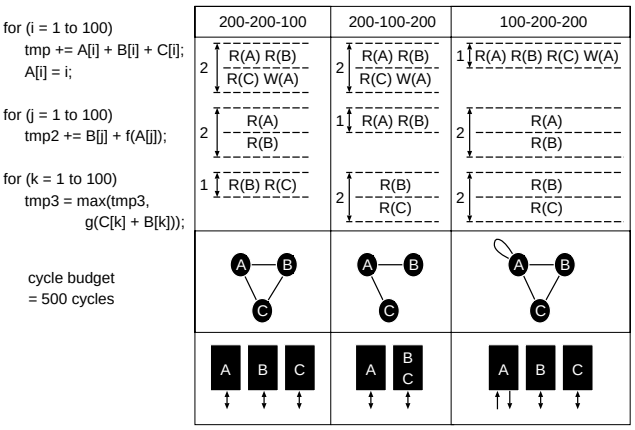

Figure 2. The distribution over loops has a big impact on the memory architecture.

5.2 Global optimum for ECG

A global optimization over all blocks is needed to obtain the global optimal conflict graph. Ordering the memory accesses on a block per block basis will result in a poor global result. The local solution of one block will typically not match the (local) solutions found in other blocks. Together, the local optima will then sum up to an expensive global solution (see Figure 3).

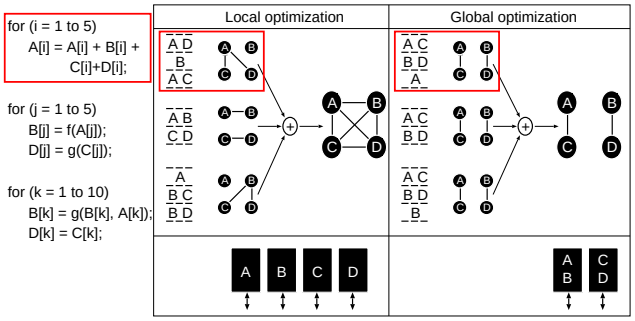

Figure 3. Global optimum instead of local optima.

Solving the problem locally per block will lead to different conflict graphs. The total application has one memory architecture and therefore one global conflict graph only. The memory architecture cannot change from one block to the next. Therefore, all the block (local) conflicts graphs should be added to one single global conflict graph. A typical conflict mismatch is shown in the left side Figure 3. A fully connected global graph is the result, requiring four memories.

Ordering the memory accesses with a global view can potentially "reuse" conflicts over different blocks. When the different blocks use the same conflicts, as shown at the right hand side of Figure 3, the global conflict graph and the resulting memory architecture are much cheaper. Again, the academic example shows the essence of the problem. However, the real problem is much more complex.

6 Incremental SCBD

For solving the storage cycle budget distribution over multiple blocks we have placed the flat-graph solver (described in Section 4) in a loop, iterating over all the blocks. As motivated in Subsection 5.2 as such, this leads to a poor global optimum. This is solved by suggesting reusable conflicts to the flat-graph solver. Moreover, additional constraints are put forward to steer the process.

The incremental SCBD reduces the global cycle budget every step until the target cycle budget is met. Initially, the memory access ordering is sequential. Therefore every block can be ordered without any conflicts. During the iteration over the blocks, the cycle budget is made smaller for every individual block. Gradually, more conflicts will have to be introduced. The SCBD approach decides which block(s) are reduced in local cycle budget and so which conflicts are added globally. Finally, after multiple steps of decreasing the budgets for the blocks, the global cycle budget is met and a global conflict graph is produced. This is less trivial than it looks, as shown next.

The proposed SCBD algorithm initializes with a sequential ordering (see Figure 4). Every memory access has its own time slot. A block containing X memory accesses will have an initial cycle budget of X cycles (assuming one cycle per memory access). Due to this type of ordering, the global conflict graph will not contain any conflicts.

Initialisation:

```
for (all blocks)
  Use sequential ordering

The found ECG will contain NO conflicts
```

Iteration:

```
While (target cycle budget > cycle budget)  // step
  for (all blocks)
    reduce cycle budget of block
    return (some) ordering freedom
    execute flat-graph optimizer
  update one or multiple blocks (based on gain/cost)
```

Figure 4. Basic algorithm for incremental SCBD.

In the successive steps of the algorithm the global budget will shrink. Every step, (at least) one of the blocks will reduce in length. The reduction of the block is at least one cycle. However, depending on the number of iterations of the concerning block, the impact on the global budget is much bigger. The global conflict cost change is calculated in the case of the block reduction. But the reduction is not approved yet. The block(s) having the biggest gain (based on a change in total cycle budget and/or change in conflict

cost) is actually reduced in size. All the other ordering results in this step are discarded. The cycle budget reduction is continued until the target cycle budget is reached. Due to the block cycle budget reduction, additional conflicts arise. Note that this basic algorithm is greedy, since only a single path is explored to reach the target cycle budget.

The traversal of the cycle search space can be made less greedy however. At every step, multiple reduction possibilities exist. Instead of discarding non-selected block ordering information (as proposed in the previous paragraph), these can be selected and explored further. Different block reductions (paths) and finally the entire solution tree can be built. An example of such one additional exploration is shown by the dotted line in Figure 6. Many of the branches will be equivalent to the already found "greedy solutions". Due to this property, the exploration will not explode but still extra solutions can be found. Moreover, a lower bound can also cut off some of the potential branching paths. Note however, due to the heuristics in the flat graph solver, the solution may be different even though the distribution of cycles over the blocks is equal. In this way, the longer the tool will run, the more and (maybe) better solutions can be found.

The incremental nature is further explained in Figure 5. The source code is shown left. The first loop contains five memory accesses and has five iterations. Therefore, in total 25 cycles are spent in *loop-i* when executed sequentially. Similarly, 20 cycles in *loop-j* and 50 cycles in *loop-k*. In total 95 cycles are spent for the entire algorithm. In this example, *loop-i* is selected in the first step and reduced from five to four cycles, causing a total cycle budget reduction from 95 to 90. In the next two steps, *loop-k* is reduced, leading to a total cycle budget from first 80 and then 70. In the last step, the target budget is met and the algorithm ends.

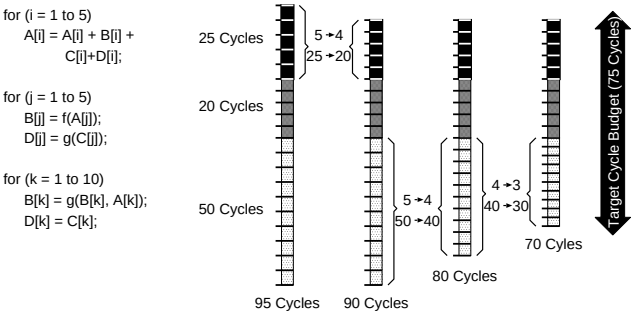

Figure 5. Graphical representation of incremental SCBD.

To improve the global result and to avoid a long execution time and instable behavior, two new inputs have to be entered to the flat-graph scheduler. First of all, a list of reusable conflicts is specified. The internal cost function is adapted to (re)use conflicts which are already used by other blocks if possible. Second, the ordering freedom is limited. The returned ordering freedom to a block is based on the final ordering of the previous step[1].

The algorithm is sped up further by reusing ordering results which did not change. Since much ordering information is discarded in a step, this does not mean it is useless. By keeping track of which blocks have to be rescheduled, the tool execution time can be decreased drastically. This happens especially in large applications containing many independent blocks.

Currently, the flat flow graph technique (see Section 4) used has already proven its value in the past. But its speed becomes unacceptable large for huge loop bodies. Research is in progress to use dynamic programming and different type of schedulers to drastically speedup the original techinique [12].

We have built a prototype tool which incorporates all these techniques, to avoid the tedious labour for a designer. An on-line demo and down-loadable (application restricted) executable are available on the web:
http://www.imec.be/acropolis/

7 Experimental SCBD results

The new prototype tool has been applied to drivers from multiple application domains to prove the effectiveness, as demonstrated by results in other recent papers [2, 3, 4, 17]. In these papers, the actual technique underlying the SCBD exploration has however not yet discussed. Here we will analyse the results and the detailed evolution of the SCBD tool for the Binary Tree Predictive Coder driver only. The experiment clearly shows the tradeoff between memory organisation cost (power and area) and the memory subsystem cycle budget. All the results are obtained within reasonable tool execution time (several minutes) on a Pentium II-400. A Motorola library memory model is used to estimate the on-chip memory cost (see [6]). For the off-chip components, we have used an EDO DRAM series of Siemens.

Binary Tree Predictive Coding (BTPC) [15] is a lossless or lossy image compression algorithm based on multi resolution. The image is successively split into a high resolution image and a low resolution quarter image, where the low-resolution image is split up further. The pixels in the high-resolution image are predicted based on patterns in the neighboring pixels. The remaining error is then expected to achieve high compression ratios with a Huffman coder. The power numbers in this section are based on real memory models.

Figure 6 shows the tradeoff for the complete cycle budget range. This is obtained by letting the tool explore the cycle budget starting from the fully sequential budget, and then progress through the most interesting memory organi-

[1]The memory access is scheduled between the ASAP and ALAP time. Both the ASAP and ALAP are put close to the location of the previous ordering.

sations (from a cost point of view) to reduce the cycle budget for multiple differently optimized implementations [4]. The number of allocated on chip memories is four for the entire graph shown in this figure. In the sequential budget (about 18M cycles), only single-port memories are employed. When a dual-ported memory has to be added, a clear discontinuity is present in the energy function. In order to reduce the budget below 8M cycles, dual-ported memories are needed though. The allocation of two dual-ported memories allows to decrease the cycle budget up to the critical path (6.5M cycles). The worst case needed "image" bandwidth can be guaranteed by inserting three intermediate on-chip memories (two dual port and one single port). These three intermediate memories can deliver up to five pixels per memory cycle.

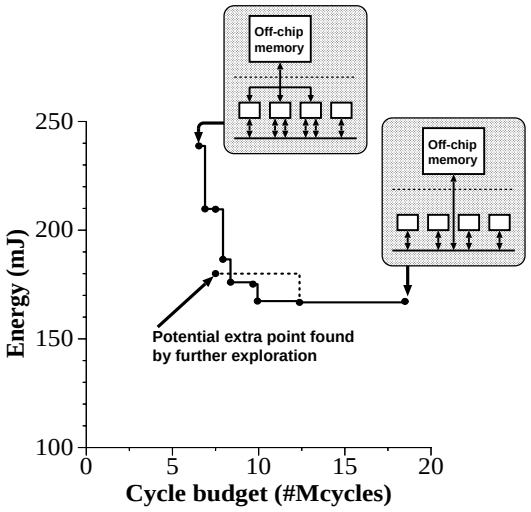

Figure 6. Pareto curve for Binary Tree Predictive Coder.

The cyle distribution over the eight loops of the BTPC application is done incrementally by the tool. The evolution of this distribution is shown in Figure 7. At every step, the differently shaded bars represent the consumed cycle budget of every loop ("loop body time" $\times$ "number of loop iterations"). The lowest gray part represents *loop-1*, the next white part represents *loop-2*, the next one *loop-3* etc.. As can be seen in the figure, *loop-1* is dominant and consumes nearly 50% of the overall time in the fully sequential case (step-0). In addition, *loop-4* is so small that it only appears as a thin line. The dotted line and solid line represent respectively the estimated cost (internally estimated in SCBD without generating a memory organisation) and the actual power cost after memory allocation and assignment.

In the progressing steps of SCBD, the cycle budget is reduced and conflicts are added gradually. Step-0 is the fully sequential budget containing no conflicts. Therefore, the estimated cost is zero. However, the memory architecture will (of course) consume some power. In the first step *loop-2, 3* and *8* are reduced in cycle budget, decreasing also the global cycle budget just below 20M cycles. The (few) added conflicts increase the cost estimate. The actual power does not increase because the added conflict does not enforce changes in the optimal signal to memory assignment. The actual power cost does not increase until step-11. Then the cycle budget is reduced by 40% already. Due to an added selfconflict in step-16, both the estimated cost and actual cost make a large jump. For step-24 and step-25 the memory architecture constraints are demanding a 4-port memory. For the used memory library this is not feasable. Therefore, no valid solution exists any longer.

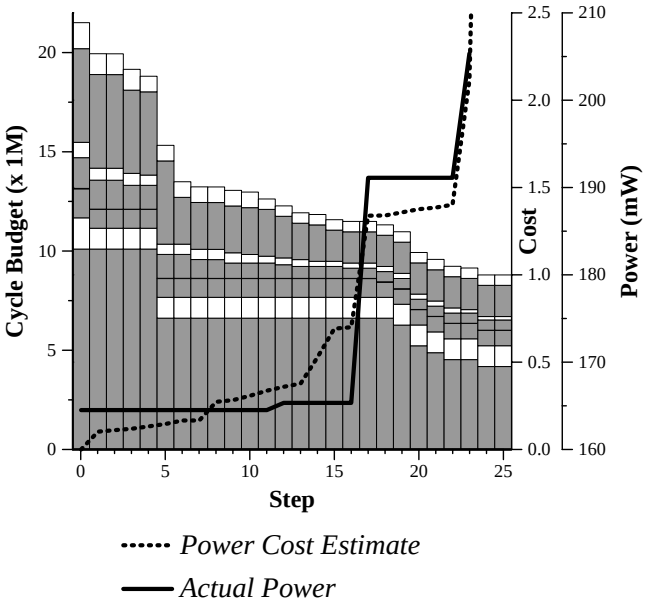

Figure 7. The distribution of cycles over the loops.

The tool feedback is more then a memory architecture alone. Optimization, for further cycle budget or cost, reduction can be based on the found schedule and conflict graph. A scheduled flow graph of a single loop is given in Figure 8. Also the global conflict graph of this step is given in Figure 9. The global conflict graph contains the conflict of the scheduled loop and of all the other loops. So in timeslot-2 the signals *in* and *glob_freq1* are scheduled together, and therefore it appears in the conflict graph. The gray nodes and edges in Figure 8 show the critical path. Indeed, the dataflow of the critical needs to be broken to reduce cycle budget further. This is typically done with software pipelining techniques which are not discussed in this paper. Another approach to reduce the cycle budget further is partially loop unrolling. Then, a large parallelism becomes available. Note that this comes with a potential very high cost in both required bandwidth and code size. The cost of the memory architecture is reflected in the conflict graph. By analyzing the conflict graph at every step it can be located where a cost increase occurs and why. Counter measures can be taken in the form of basic group matching [9] and inserting

intermediate arrays [10].

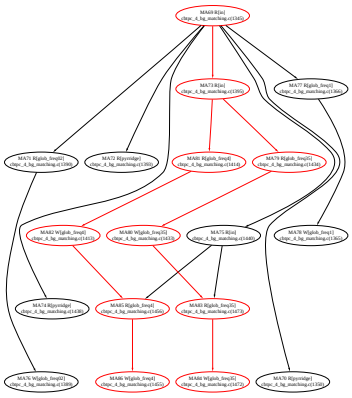

Figure 8. Scheduled flow graph of one loop in step 23.

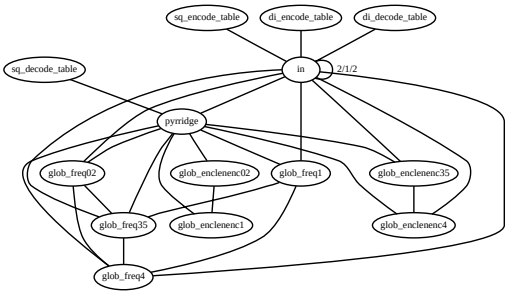

Figure 9. The conflict graph for step 23.

These large performance gains are only possible at moderately low cost when the memory architecture is designed carefully. At step-23 the cycle budget is reduced with 60% at a cost of 25% more power consumption, 5% more area and more complex interconnect compared to the fully sequential solution (step-0).

This experiment also proves the high usefulness of the tool. All these results can only be generated effectively by the introduction of our new approach and tool. It is clear that manual design would never allow to explore such a huge search space.

8 Conclusions

In the past, our Flow Graph Balancing technique and tool were applicable for flat flow graphs only. All real-life application, however, contain loops and a function hierarchy. So the tool could only be applied relatively locally on a loop per loop basis where the designer has to distribute the global cycle budget over the loops. In this paper, the existing methodology is extended towards hierarchy support. With the current extension we tackle the two most urgent problems. First, the memory cycle budget distribution across blocks is automated. And second, a global optimum for the entire application is found and not a local optimum for every block separately. This paper has shown the effectiveness of our new incremental approach on a real-life demonstrator. The power, area and timing information obtained from the tool are very useful especially at the system level. Clear tradeoffs can be made on a higher level to steer code transformations using these estimations.

References

[1] F.Balasa, F.Catthoor, H.De Man, "Dataflow-driven memory allocation for multi-dimensional processing systems", *Proceedings IEEE International Conference on Computer Aided Design*, San Jose CA, Nov. 1994.

[2] E.Brockmeyer, J.D'Eer, N.Busa', F.Catthoor, P.Lippens, J.Huiskens, "Code transformations for reduced data transfer and storage in low power realization of DAB synchro core", Patmos'99, Kos, Greece, Oct 6-8, 1999.

[3] E.Brockmeyer, S.Wuytack, A.Vandecappelle, F.Catthoor, "Low power storage for hierarchical graphs", *Proc. 3rd ACM/IEEE Design Automation Test in Europe Conf.*, Paris, France, pp., April 2000.

[4] E.Brockmeyer, A.Vandecappelle, F.Catthoor, "Systematic Cycle budget versus System Power Trade-off: a New Perspective on System Exploration of Real-time Data-dominated Applications", to appear in *Proc. Intnl. Symp. on Low Power Design*, Rapallo, Italy, July 25-27, 2000.

[5] F.Catthoor, S.Wuytack, E.De Greef, F.Franssen, L.Nachtergaele. H.De Man, "System-level transformations for low power data transfer and storage", in paper collection on "Low power CMOS design" (eds. A.Chandrakasan, R.Brodersen), IEEE Press, pp.609-618, 1998.

[6] F.Catthoor, S.Wuytack, E.De Greef, F.Balasa, L.Nachtergaele, A.Vandecappelle, "Custom Memory Management Methodology – Exploration of Memory Organisation for Embedded Multimedia System Design", ISBN 0-7923-8288-9, Kluwer Acad. Publ., Boston, 1998.

[7] F.Catthoor, K.Danckaert, C.Kulkarni, T.Omnes, "Data transfer and storage architecture issues and exploration in multimedia processors", book chapter in "Programmable Digital Signal Processors: Architecture, Programming, and Applications" (ed. Y.H.Yu), Marcel Dekker, Inc., New York, 2000.

[8] E.De Greef, F.Catthoor, H.De Man, "Memory Size Reduction through Storage Order Optimization for Embedded Parallel Multimedia Applications", special issue on "Parallel Processing and Multi-media" (ed. A.Krikelis), in *Parallel Computing* Elsevier, Vol.23, No.12, Dec. 1997.

[9] P.Ellervee, M.Miranda, F.Catthoor, A.Hemani, "Exploiting data transfer locality in memory mapping", *Proc. 25th EuroMicro Conf.*, Milan, Italy, pp.14-21, Sep. 1999.

[10] J.P.Diguet, S.Wuytack, F.Catthoor, H.De Man, "Formalized methodology for data reuse exploration in hierarchical memory mappings", *Proc. IEEE Intnl. Symp. on Low Power Design*, Monterey, pp.30-35, Aug. 1997.

[11] P.Lippens, J.van Meerbergen, W.Verhaegh, A.van der Werf, "Allocation of Multiport Memories for Hierarchical Data Streams", *Proc. IEEE Inter. Conf. on Computer-Aided Design*, pp.728-735, Santa Clara, Nov. 1993.

[12] T.Omnes, T.Franzetti, F.Catthoor, "Interactive algorithms for minimizing bandwidth in high throughput telecom and multimedia", accepted for *Proc. 37th ACM/IEEE Design Automation Conf.*, Los Angeles CA, pp., June 2000.

[13] P.Paulin, J.Knight, "Force-directed scheduling for the behavioral synthesis of ASIC's", *IEEE Trans. on CAD*, Vol.8, No.6, pp.661-679, June 1989.

[14] N.Passos, E.Sha, "Push-up scheduling: optimal polynomial-time resource constrained scheduling for multi-dimensional applications", *Proc. IEEE Inter. Conf. on Computer-Aided Design*, San Jose CA, pp.588-591, Nov. 1995.

[15] J. Robinson, Efficient general-purpose image compression with binary tree predictive coding. *IEEE Trans. on Image Processing*, 6(4):601–608, Apr. 1997.

[16] L.Stok, "Data path synthesis", *INTEGRATION, the VLSI journal*, Vol.18, pp.1-71, June 1994.

[17] A.Vandecappelle, M.Miranda, E.Brockmeyer F.Catthoor, D.Verkest, "Global Multimedia System Design Exploration using Accurate Memory Organization Feedback" *Proc. 36th ACM/IEEE Design Automation Conf.*, June 1999.

[18] W.Verhaegh, P.Lippens, E.Aarts, J.Korst, J.van Meerbergen, A.van der Werf, "Improved Force-Directed Scheduling in High-Throughput Digital Signal Processing", *IEEE Transactions on CAD and Systems*, Vol.14, No.8, Aug. 1995.

[19] W.Verhaegh, "Multidimensional Periodic Scheduling", *Ph.D. dissertation*, Eindhoven University of Technology, Oct. 1995.

[20] S.Wuytack, F.Catthoor, F.Franssen, L.Nachtergaele, H.De Man, "Global communication and memory optimizing transformations for low power systems", *IEEE Intnl. Worksh. on Low Power Design*, Napa CA, pp.203-208, Apr. 1994.

[21] S.Wuytack, J.P.Diguet, F.Catthoor, H.De Man, "Formalized methodology for data reuse exploration for low-power hierarchical memory mappings", *IEEE Trans. on VLSI Systems*, Vol.6, No.4, pp.529-537, Dec. 1998.

[22] S.Wuytack, F.Catthoor, G.De Jong, H.De Man, "Minimizing the Required Memory Bandwidth in VLSI System Realizations", *IEEE Trans. on VLSI Systems*, Vol.7, No. 4, pp.433-441, Dec. 1999.