Using Weighted Graphs for Fast Architecture Exploration

Peeter Ellervee, Tarmo Klaar*, Margus Kruus, Kalle Tammemäe

Dept. of Computer Engineering, Tallinn Technical University, Raja 15, 12618 Tallinn, Estonia
**MicroLink Computer Ltd., Estonia*
e-mail: lrv@cc.ttu.ee, tarmo@microlink.ee, {kruus nalle}@cc.ttu.ee

Abstract -- Architecture exploration in the SoC era is one of the most crucial optimization tasks. The pressure from the market, the enormous number of possible solutions and the computationally hard nature of the optimization tasks is the pushing force behind the quest for fast and good enough optimization algorithms. In this paper, some hardware optimization tasks are represented as weighted graph coloring and clustering problems. Fast and simple heuristics have been developed that find close to optimal solutions.

1. Introduction

The latest developments in the electronic industry, namely the possibility to create and connect billions of transistors on a single chip, require also new design and synthesis methodologies. One of the most important features for the new methodologies is the need to perform architecture exploration at various levels of abstractions. Designing a whole system on a single chip (System-on-a-Chip or SoC) requires also optimization algorithms that take into account effects from different abstraction levels, and/or take into account effects from other optimization tasks.

This leads to three important characteristics an optimization algorithm should have - a) the algorithm should be fast enough to search through the ever increasing solution space; b) the algorithm should be efficient enough to lead to a solution that differs from the optimal solution as little as possible; and c) the algorithm should allow to define complex relationships between system components.

Graph optimization algorithms have been used for various optimization tasks at different abstraction levels. Classically, a larger optimization task is divided into sub-tasks and each of these sub-tasks is solved as a conflict graph coloring task; or a compatibility graph clustering task.

For instance, the tasks at register-transfer level that are often mapped to graph coloring tasks are allocation and binding tasks. These tasks, mapped onto conflict graph coloring or compatibility graph partitioning tasks, have been studied by many authors and various solutions have been proposed to unify them. All these optimization tasks are intractable for real-life size examples. Thus, different heuristics that generate solutions fast without guaranteeing optimality have been used widely (see, e.g., [1,2]).

In this paper, few examples how to use the fast optimization algorithms for SoC architecture exploration, described in [3], are presented. The original algorithms were created to solve only the hardware allocation and binding tasks in a unified manner. The optimization task was mapped onto weighted conflict graph coloring task. Initially, four simple greedy heuristics were used. Later the heuristics have been extended in two ways:

- selection criteria have been extended to widen search space with a goal to improve results (with the penalty of speed reduction, of course); and
- cost functions have been extended to cover multiple physical dimensions, e.g., area, power.

These extensions allowed to map some other optimization tasks, e.g., memory optimization, onto weighted graph coloring or partitioning/clustering tasks.

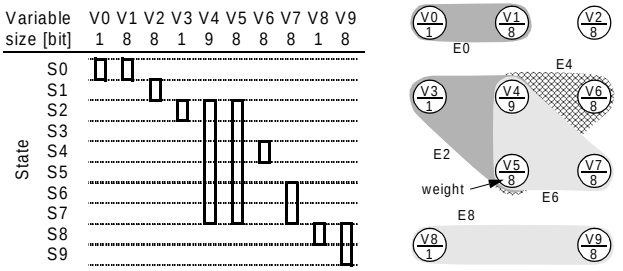
2. Optimization tasks

In this section, the mapping of five hardware synthesis tasks onto weighted hyper-graph optimization tasks is described. The first three tasks - functional unit and register allocation and binding at register-transfer level; and memory allocation and binding at functional level - are solved as weighted conflict graph coloring tasks. The last two tasks - system task allocation and state machine decomposition - are mapped onto weighted compatibility graph clustering tasks.

The nodes of a conflict graph represent behavioral entities (variable, operation). Hyper-edges give the conflict relation, i.e., there is an edge when operations can not share the same resource, e.g., operations are executed in paral-

lel. All nodes are associated with one or more weights that are used by the cost function. The weights are assigned during the graph construction and represent operational characteristics, e.g., bit-width, access frequency.

Conflict graphs are built in a similar manner for all three coloring tasks - the underlying control-flow or data-flow is analyzed to find operations that are executed at the same time. For register allocation, for instance, the life-time intervals of variables are collected. Variables with covering lifetimes define the conflicting nodes of the graph. An example of lifetime intervals of an algorithm and corresponding conflict hyper-graph are shown in Figure 1. This allows a better estimation of access frequencies (there is exactly one edge per sate), especially when each hyper-edge has a weight associated with the profiling information. The conflict graph construction and the coloring tasks are described in [3].



a) lifetime intervals of variables b) conflict hyper-graph

Figure 1. Lifetime intervals and corresponding conflict graph.

2.1. Register and functional unit allocation / binding

The conflict graphs used for register and functional unit allocation and binding tasks are typically built by analyzing a scheduled control and data flow graph (CDFG) of an algorithm. An example CDFG is shown in Figure 2.a). Variables **x** and **y** are stored at the same time and thus create an hyper-edge. The register that is used to store variable **z** can be used to store, in principle, also either **x** or **z**. However, when **z** share the register with **x** (instead of **y**) increase both in area and power consumption should be expected. The area increase is caused by the need to use 32-bit multiplexer instead of 8-bit multiplexer. The increase in power consumption is caused by the triggering of extra 24 flip-flops when storing variable **z**.

Cost functions for area (node cost - bit-width - w_i):

- area of a single register - $area_{reg}(w_i)$ - gates for w_i-bit wide register;
- area for the shared register (i and j) - $area_{reg}(max(w_i,w_j))+area_{mux}(2,max(w_i,w_j))$ - area of the register to store the largest of the variables, plus area of 2-input multiplexer.

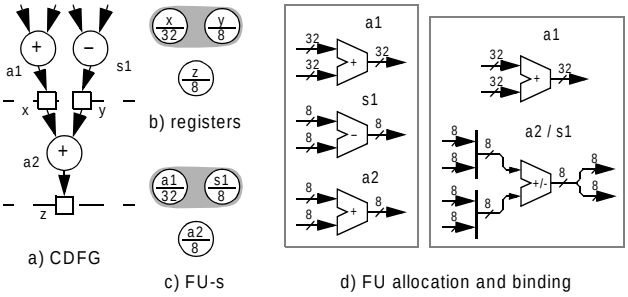


Figure 2. CDFG and corresponding conflict graphs

Cost functions for power consumption (node cost - access frequency - f_i):

- power consumption of a single register - $power_{reg}(w_i,f_i)$;
- power consumption for the shared register (i and j) - $power_{reg}(max(w_i,w_j),f_i+f_j) + power_{mux}(2,max(w_i,w_j),f_i+f_j)$.

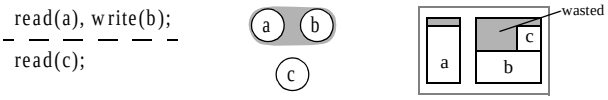
The conflict graph used for functional unit (FU) allocation and binding task is built and used, in principle, not unlike for register allocation and binding. Figure 2.c) depicts the conflict graph that corresponds to the CDFG in Figure 2.a). Additionally, the cost functions should take into account the type of the FU. For instance, in Figure 2.d), two different allocation and binding results of the conflict graph are shown. In the first case, separate adder and subtracter are used to implement operations **a2** and **s1**. In the second case, both operations have been implemented by using an adder-subtracter (plus two multiplexers).

It should be noted that the actual parameters for the cost functions are technology dependent. For instance, cost of a multiplexer in LSI-10K technology can be approximated as: $area_{mux}(n,w) = w\,(0.083n^2+1.49n+0.154)$; where n is the number of data inputs and w is the width in bits.

2.2. Memory allocation and binding

Profiling memory accesses at functional level allows a significant reduce in complexity of the overall design task. Also, reduced task complexity allows wider exploration of different memory configurations, i.e., to select the most suitable memory architecture. The conflict graph used for memory allocation and binding task is typically built by analyzing the ordered sequence of an algorithm. An example sequence is shown in Figure 3.a). Arrays ‘a’ and ‘b’ are accessed at the same time and thus create an hyper-edge. The resulting conflict graph (Figure 3.b) allows three legal mappings of arrays onto physical memories (allocation and binding) as shown in Figure 3.c). The most suitable solution depends on many factors - wasted memory area, number of buses and address generators, access frequencies, etc. [4]

The following cost functions allow to estimate the memory area (without bus and address generator costs) where node cost - word-width (w_i), number of words (n_i):



a) sequence b) conflict graph c) legal mapping

Figure 3. Memory allocation as a graph coloring task.

- area of a single memory - $area_{mem}(w_i,n_i)$ - n_i*w_i bits, plus technology dependent overhead;
- area for the shared memory (*i* and *j*) - $area_{mem}(max(w_i,w_j),n_i+n_j)$.

The cost function for power consumption can be defined in a similar manner (additional node cost - read and write frequencies - f_i^r,f_i^w):

- power consumption of a single memory - $power_{mem}(area_{mem}(w_i,n_i),f_i^r,f_i^w)$;
- power consumption for the shared memory (*i* and *j*) - $power_{mem}(area_{mem}(max(w_i,w_j),n_i+n_j),f_i^r+f_j^r,f_i^w+f_j^w)$.

2.3. System task and state machine partitioning

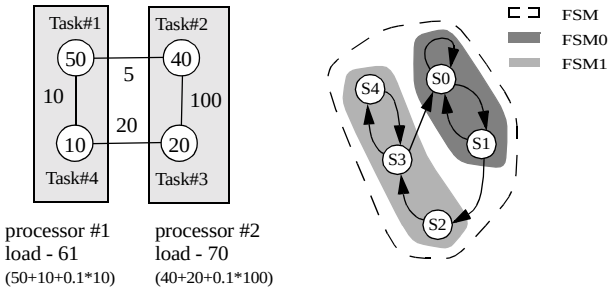
The system task allocation and state machine decomposition are mapped onto weighted compatibility graph clustering tasks. This mapping exploits an interesting feature of weighted graphs that is essentially an extension of duality of graph coloring and partitioning tasks [1]. Typically, a weight on an edge represents how close two connected nodes are. At the same time, cost functions can be defined in such a way that a large enough weight acts as a very large penalty when trying to group two nodes that are associated with such an edge. The optimization tool interprets both conflict and compatibility graphs in the same manner. The only difference is that edges of conflict graphs have fixed weights that are equal to infinity [3].

Figure 4.a) depicts an example of representing a set of communicating tasks as a weighted graph. Weights of edges represent the amount of data to be transferred between tasks and weights of nodes represent the number of calculations a task must perform. Two example cost functions can be defined as follows:

- load of a cluster (processor) - sum of node weights, i.e., all calculations, plus weighted sum of internal edge weights, i.e., all in-cluster data transfers; and
- communication overhead - weighted sum of edge weights between clusters, i.e., data transfers between processors.

The cost functions can be more complex, of course. The importance is that these functions should be able to give fast estimates when deciding one or another mapping of tasks onto processors [5].

Similarly, activity in state machines can be represented as a weighted graph where edge weights represent transition probabilities and node weights represent state operations (Figure 4.b). Such model is very useful when partitioning state machine to save power and/or energy [6].



a) system task mapping b) state machine partitioning

Figure 4. Compatibility graph clustering tasks.

3. Heuristic graph optimization algorithms

The heuristic algorithms, described below, are based on the greedy constructive approach where colors (clusters) are assigned to nodes sequentially. The greedy approach is the simplest amongst all of the approaches - it is both easy to implement and it is computationally inexpensive.

For all allocation and binding tasks, a node in the conflict graph corresponds to a behavioral entity (variable, operation, array), and a color corresponds to a resource (register, functional unit, memory) used for its implementation. Cost of the existing coloring is evaluated at every step and the cheapest of two alternatives - using an existing color or creating a new one - is used. All four heuristics use the same core algorithm and they differ only how the nodes are ordered for coloring. The core algorithm and extensions are described in [3].

Figure 5. depicts some of the search trees to illustrate how a solution may be generated (the breadth-first-search approach has been assumed). The simple static approaches check and store only a single solution at every step - they are very fast but too often get stuck into local optima (Figure 5.a). The dynamic greedy approach checks all possibilities but stores also a single solution (Figure 5.b). The extensions were used to overcome the main drawbacks of the simple greedy approaches. They expand both checking and storing in such a way that more than one possible solution is checked and/or stored at every iteration (Figure 5.c).

It should be noted that both extensions increase also the computation time. The increase depends linearly on multiplication of numbers of the extra checking and of the stored solutions. Thus, designer can make trade-offs between design quality and time.

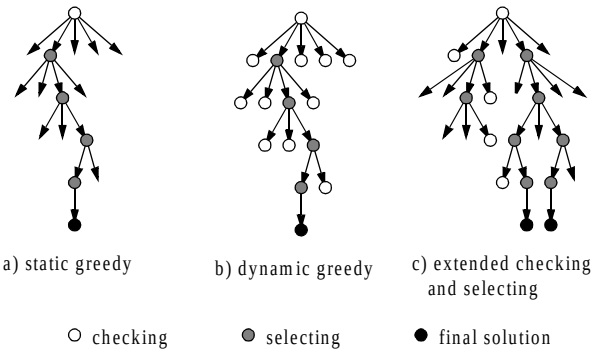


Figure 5. Search trees.

4. Experiments

More than 200 coloring tasks from real-life designs were used to compare the heuristic algorithms (see Table 1.). The largest designs had 222 functional units that were bound into 92 components, and 91 variables bound into 29 registers. It should be noted that the algorithm H3 was used with multiple trials, and the best of the bindings was selected. The number of trials was set to the number of nodes, but in many cases the best result, or one of the best, was achieved with the first trial. Only for 58 designs the results from different heuristics were different (and different also from the global optima). For the rest of the designs all heuristic algorithms obtained the optima.

Table 1. Efficiency of different heuristics

Selection criterion	Best / Minimal	Average penalty	Maximum penalty	Average time
H1 (largest)	31 / 18	2.8%	8.0%	2.8”
H2 (smallest)	21 / 13	3.8%	10.4%	2.9”
H3 (random)	73 / 49	1.2%	5.6%	31.5”
H4 (dynamic)	10 / 5	4.1%	13.3%	185”

The column “Best/Minimal” shows how many of the bindings by that algorithm were also the overall best ones and how many of the bindings were also globally minimal. The next two columns show how many percents the cost of the resulted bindings differ from the global minimum. The last column shows the average coloring time of real-life graphs with 50 to 100 nodes.

The coloring results of 150 random graphs gave comparable results - the average penalty was 1% for H3 and 3-4% for the others (the maximum penalty was 3% and 10-12%, correspondingly).

The effects of extensions were analyzed by coloring more than 120 random graphs (10 to 40 nodes) with one to eight nodes checked and one to eight partial solutions stored at every iteration. Results for some interesting cases are presented in Figure 2. It should be noted that although in average the results were always improved, in very few cases the extended heuristics gave slightly inferior results, a topic for future research.

Table 2. Efficiency of extended heuristics

Selection criterion	Best / Minimal	Average penalty	Maximum penalty	Average time
H1 (largest)	85 / 85	1.0%	10.2%	0.1”
check 4, store 4	93 / 93	0.6%	10.8%	0.9”
H2 (smallest)	90 / 89	1.2%	16.9%	0.1”
check 8, store 4	91 / 91	0.9%	10.8%	1.9”
H3 (random)	109 / 107	0.3%	7.5%	3.1”
check 8, store 1	112 / 109	0.2%	3.8%	35”

5. Conclusion

This paper has shown that fast and simple greedy optimization algorithms give good results, thus allowing to widen search of different architectural solutions. Several optimization tasks have been mapped onto weighted graph coloring and/or clustering tasks. Based on the results from experiments with more than 400 optimization tasks (real-life tasks included), it can be stated that the used heuristics will lead to good enough results in a matter of seconds. In a significant number of cases they may also lead to the global optimum.

This work was supported in part by the Estonian Science Foundation Grants no. 4294 and 5141.

References

[1] G. De Micheli, “Synthesis and Optimization of Digital Circuits,” McGraw-Hill, Inc., 1994.

[2] S.-Y. Yuan, S.-Y. Kuo, “A New Technique for Optimization Problems in Graph Theory,” IEEE Trans. on Computer, Vol. 47, No. 2, pp. 190-196, Feb. 1998.

[3] P. Ellervee, T. Klaar, “Using Weighted Graph Coloring Heuristics for Architecture Exploration,” The 19th NORCHIP Conference, pp. 161-166, Stockholm, Nov. 2001.

[4] 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,” Kluwer, 1998.

[5] A. Hemani, A. Jantch, S. Kumar, A. Postula, J. Öberg, M. Millberg, D. Lindqvist, “Network on a Chip: An architecture for billion transistor area”, The 18th NORCHIP Conference, pp. 166-173, Turku, Nov. 2000.

[6] B. Oelmann, K. Tammemäe, M. Kruus, M. O’Nils, “Automatic FSM Synthesis for Low-Power Mixed Synchronous/ Asynchronous Implementation,” Special Issue on Low Power System Design Issues of the VLSI Design Journal, Gordon and Beach Science Publ., Vol. 12, No. 2, pp. 167-186, 2001.