Color Permutation: an Iterative Algorithm for Memory Packing

Jianwen Zhu
Edward S. Rogers Sr.
Department of Electrical and Computer Engineering
University of Toronto, Ontario M5S 3G4, Canada
jzhu@eecg.toronto.edu

Abstract

It is predicted that 70% of the silicon real-estate will be occupied by memories in future system-on-chips. The minimization of on-chip memory hence becomes increasingly important for cost, performance and energy consumption. In this paper, we present a reasonably fast algorithm based on iterative improvement, which packs a large number of memory blocks into a minimum-size address space. The efficiency of the algorithm is achieved by two new techniques. First, in order to evaluate each solution in linear time, we propose a new algorithm based on the acyclic orientation of the memory conflict graph. Second, we propose a novel representation of the solution which effectively compresses the potentially infinite solution space to a finite value of $n!$, where n is the number of vertices in th memory conflict graph. Furthermore, if a near-optimal solution is satisfactory, this value can be dramatically reduced to $\chi!$, where $\chi!$ is the chromatic number of the memory conflict graph. Experiments show that consistent improvement over scalar method by 30% can be achieved.

1 Introduction

Today's telecommunication and consumer electronics applications demand computational power that can be met only by integrating more and more hardware components. Given that such applications typically buffer and process a large amount of data, the interface between logic and memory tends to become the performance bottleneck. While memories employing advanced signaling techniques such as Rambus memories are emerging to alleviate the problem, it is often simpler and faster to integrate memory and logic on a single chip. It is hence not surprising to find on-chip memories to occupy a larger portion of silicon area than logic does in the future systems-on-chips. While traditional CAD has devoted to the minimization of logic area in order to reduce manufacturing cost, which exponentially depends on the die size, the interest in the minimization of memory size, has emerged only recently.

Naturally, one can achieve memory saving by overlapping the address space of distinct memory blocks by analyzing their lifetimes. The data analysis techniques, be it array-based or pointer-based [13], establish the conflict relationship between the life time of program memory blocks (or even subblocks). The problem of mapping memory blocks to addresses which minimize the total size of the address space, while honoring the conflict relation, remains to be solved. Previous methods either use a naive extension of the scalar register allocation algorithms, which produce suboptimal results; or use a heuristic algorithm of cubical complexity, yet with no guarantee of optimality. In this paper, we develop a new algorithm under the classical framework of iterative improvement, where either a greedy or simulated annealing strategy can be used. The contribution of this algorithm is three-fold: First, we find that an acyclic orientation of the undirected conflict graph leads to a linear algorithm for memory packing and therefore is perfect for solution evaluation. Second, we are able to discover a finite solution space that is *P-admissible* in the sense that an optimal solution is guaranteed to be included. This solution space has a size of $n!$, where n is the number of the vertices. Third, we show that if the P-admissibility can be relaxed, we can dramatically reduce the size of the solution space to $\chi!$, where χ is the chromatic number of the conflict graph, thereby dramatically reduce the time of convergence. Fortunately, experiments show that near-optimal solutions can be found within this solution space.

The rest of the paper is organized as follows: In Section 2, we discuss related work. In Section 3, we formally define the problem. In Section 4 we present our algorithm in detail. In Section 7, we describe the evaluation methodology and show the experimental results.

2 Related Work

The storage minimization problem evolves from the scalar variable minimization problem, which manifests as the register allocation problem in the compiler community, where a heuristic-based graph coloring algorithm is found to be the most efficient in practice [3]. A simple-minded extension of the graph coloring algorithm to storage minimization leads to inferior result due to the fact that unlike registers, the sizes of the memory blocks are different.

The storage minimization problem has been attempted at the system level. For example, Bhattacharyya and Lee [1] have studied buffer minimization for the so-called synchronous dataflow SDF) programs. A SDF program models the data (memory) access explicitly using arcs between the computational actors. The buffer memory usage can be optimized by a careful schedule of actor execution.

In the high level synthesis community, [8] and [9] have studied clustering array variables into different memory blocks. [4], [10] and [11] studied the same problem with the goal of estimation in the context of system level exploration. Philip's Phideo project [5], pioneered memory architecture exploration for stream-based signal processing applications. The architecture group at UC, Irvine [7] studied the memory architecture exploration in the context of embedded processors.

The storage minimization problem for systems-on-chip has been systematically attacked at IMEC in the MATISSE project [2]. In MATISSE, a 2-stage strategy was proposed to perform the "in-place" optimization for multidimensional arrays. During the first phase, "the intra-signal windowing" is performed to interleave elements within an array. During the second phase, the "inter-signal placement" is performed to interleave arrays.

3 Problem Formulation

The input of the memory allocation problem is a set of memory blocks, as well as a *conflict* relation between these blocks, which indicate whether or not that any pair of the memory blocks can be shared, or having an overlapping memory address space. The memory block is characterized by its size, which can be any natural numbers. The conflict relation is derived by discovering the "life time" of the memory blocks using dataflow analysis, which is not the subject of this paper.

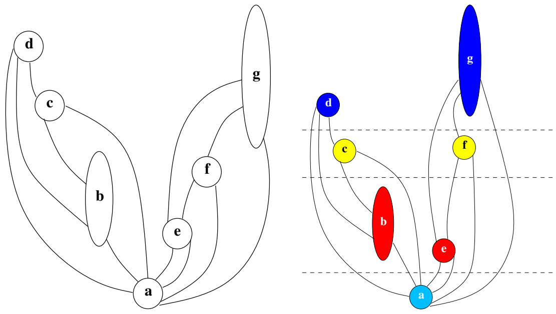
An allocation, as defined by Definition 1 is then the assignment of address location, represented by an integer, to each of the memory block, such that the conflict relation is honored.

Definition 1 *Given a set of memory blocks*[1] $V : \langle\rangle^{\mathrm{B}lock}$*, and a conflict relation* $E : \langle\rangle^{V\times V}$ *between the memory blocks, a* **memory allocation***, or a* **memory packing***, is a mapping* $\mathcal{A} : V \mapsto \mathcal{N}$*, such that* $\langle u,v\rangle \in E \rightarrow [\mathcal{A}(u), \mathcal{A}(u)+u.size] \cap [\mathcal{A}(v), \mathcal{A}(v)+v.size] = \oslash$*.*

[1] Here we use the notation $\langle\rangle^A$ to represent a power set of A, and the notation $[\,]^A$ to represent the set of all sequences over elements of A.

	a	b	c	d	e	f	g
size	1	2	1	1	1	1	3

(a) memory block sizes



(b) conflict graph (c) colored graph

Figure 1: Memory packing by coloring.

Obviously, one allocation can be better or worse than another, depending on whether or not the total memory size occupied by all memory blocks is smaller. According to Definition 2, the allocation that results in the smallest total memory size is the optimal allocation.

Definition 2 *For an allocation* $\mathcal{A} : \mathcal{V} \mapsto \mathcal{N}$*, its memory size* $||\mathcal{A}||$ *is defined to be* $max_{v\in V}\mathcal{A}(v) + v.size$*. An allocation* $\mathcal{A}_0$ *is said to be optimal if* $\forall\mathcal{A}, ||\mathcal{A}|| \geq ||\mathcal{A}_0||$*.*

4 Algorithms

In this section, we outline our proposed algorithm. For a detailed treatment, the readers are referred to [12]. To offer more insight on why we can perform better, we start by describing the use of graph coloring for memory allocation.

4.1 Graph coloring

Given a conflict graph $\langle V,E\rangle$, where V is the set of memory blocks and E is the conflict relation, a coloring algorithm assigns *colors* to each of the vertex in the graph such that no adjacent vertices have the same color. The result of coloring can be directly used to assign memory addresses by making sure that vertices with the same color will share the same memory space, while vertices with different colors will never overlap.

Example 1 *Figure 1 (a) and (b) shows a conflict graph as well as the sizes of the blocks represented by the vertices of the graph. Figure 1 (c) shows a valid coloring of the conflict graph and a strategy described above is applied to obtain a memory allocation, which has a total memory size of 7.*

It becomes immediately evident that as soon as the sizes of the memory blocks vary, the coloring-based allocation algorithm quickly degrades to suboptimal. For example, since b has a size of two, both e and f in Figure 1 (c) can share the same memory region as b, although e and f themselves shall not overlap. For the same reason, c and d should be able to share space with g, which has a size of three.

Exploiting the memory size variation is not trivial. In [2], a strategy has been employed where each memory block is attempted in a greedy fashion to be assigned an address. For each of such attempts, conflict has to be checked against the blocks that have been already assigned an address. In case of failure, another block has to be attempted. This algorithm has a cubical complexity precisely because of the amount of comparisons one has to make for conflict detection, as well as the amount of backtracking one has to perform in case of failure.

4.2 Acyclic Orientation

One approach to dramatically reduce the complexity of the cubical allocation algorithm is to carefully devise a proper *order* of address assignment so that:

- each vertex needs to be assigned only *once* (no need for backtracking);
- the conflict constraint is *implicitly* satisfied (no need for conflict checking).

We observe that such an order can be found by converting the *reflective* conflict relation into an *irreflective* partial order. In other words, converting the undirected conflict graph into a *directed acyclic graph*. With such conversion, we effectively convert the memory allocation problem into the scheduling problem, if we equate the memory space domain to the time domain, and memory block size to the delay.

Example 2 *Figure 2 (a) shows an orientation of the undirected conflict graph in Figure 1 (b). This directed graph can be "scheduled" as shown in Figure 2 (b) to obtain the memory allocation, which has a total size of 6. Note that this result is better than the one obtained in Figure 1 (c).*

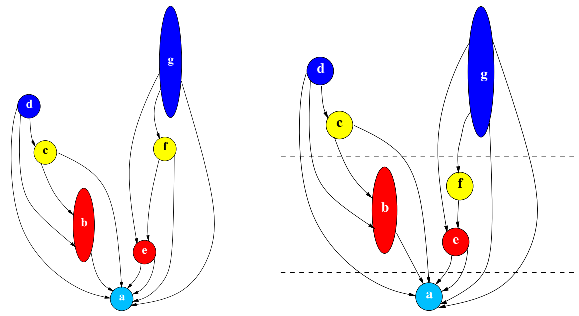
One can apply any scheduling algorithms to obtained a valid memory allocation. Theorem 1 states that the an ASAP strategy is in fact optimal for a given orientation.

Theorem 1 *Let* $g = \langle V, E\rangle \subseteq \mathrm{B}lock \times (\mathrm{B}lock \times \mathrm{B}lock)$ *be a memory conflict graph. Let* F *be an acyclic orientation. Then for any schedule* S *of* F*,* $||S|| \geq ||\mathrm{asapSchedule}(V, F)||$.

5 Vertex Permutation

Now the question is whether an acyclic orientation always exists. Theorem 2 provides a positive, constructive answer.

Definition 3 *A* permutation *of finite set* A *is a function* $P : A \mapsto \mathcal{N}$ *such that* $\forall u, v \in A. u \neq v \Rightarrow P(u) \neq P(v)$.



(a) an oriented graph (b) schedule of an oriented graph

Figure 2: Memory allocation by acyclic orientation.

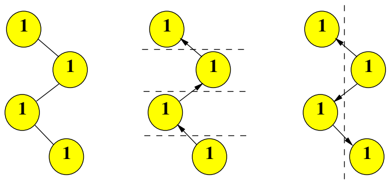


Figure 3: Good and bad orientations.

Theorem 2 *Let* $g = \langle V, E\rangle \subseteq \mathrm{B}lock \times (\mathrm{B}lock \times \mathrm{B}lock)$ *be a memory conflict graph, then for any vertex permutation* $P : V \mapsto \mathcal{N}$*, there exists an acyclic orientation* F *of* g.

What becomes crucial is whether an orientation that can lead to optimal memory allocation can be obtained. To see how the conflict graph orientation strongly affects the result of allocation, consider the example in Figure 3, where two different orientations of the same conflict graph are shown. Assume each vertex has a size of one, then the orientation at the left leads to an allocation of size 4, while the orientation at the right leads to an allocation of size 2.

Since Theorem 2 ensures that the set of all vertex permutations form a solution space of size $n!$, a heuristic search algorithm can be used to traverse the solution space, where the linear ASAP scheduling algorithm can be used to evaluate the solution. Theorem 3 and Corollary 1 ensures that an optimal solution is included in the solution space and it is therefore P-admissible. This result corresponds very well to the sequence-pair algorithm used in floorplanning [6].

Theorem 3 *Let* $g = \langle V, E\rangle \subseteq \mathrm{B}lock \times (\mathrm{B}lock \times \mathrm{B}lock)$ *be a memory conflict graph, then for any memory packing* $A : V \mapsto \mathcal{N}$*, there exists a vertex permutation* $P : V \mapsto \mathcal{N}$ *from which* A *can be derived.*

Corollary 1 *Let* $g = \langle V, E\rangle \subseteq \mathrm{B}lock \times (\mathrm{B}lock \times \mathrm{B}lock)$ *be a memory conflict graph, then there exists a vertex permutation* $P : V \mapsto \mathcal{N}$ *from which an optimal memory allocation can be derived.*

6 Color Permutation

Since $n!$ is still a large number, the search for the optimal solution can become much more efficient if the solution space can be compressed further. Our next observation is that a coloring of the conflict graph also defines an acyclic orientation.

Theorem 4 *Let $g = \langle V, E\rangle \subseteq \mathrm{B}lock \times (\mathrm{B}lock \times \mathrm{B}lock)$ be a memory conflict graph, and for any coloring $C : V \mapsto \mathcal{N}$ of g, there exists an acyclic orientation F of g.*

This leads to the strategy that a minimum coloring of the conflict graph is first found, and then different permutation of the color assignment is used to define the solution space. If we denote the chromatic number, that is, the number of color used in the minimum coloring, as χ, then the size of the solution space becomes $\chi!$, which is substantially smaller than $n!$.

Note that while the solution space is substantially compressed, it is no longer P-admissible. Fortunately, our experiments, as detailed in the next section, show that a near-optimal solution can always be found. In addition, expensive search strategies such as simulated annealing are not necessary in practice.

7 Experimental Result

We implemented the discussed algorithms in the C programming language and applied them on the DIAMCS benchmarks with randomly generated memory sizes. The result is summarized in Table 1: For each benchmark, we show its size in terms of the number of nodes and edges in the graph. We also report the allocation results for both the coloring based algorithm (color) and our proposed algorithm (perm), as well as its percentage of improvement over the coloring based algorithm. The algorithm runtime in units of milliseconds on a Ultra-5 Sun workstation with 128M of memory is also displayed.

We found that our algorithm performs on average 30% better than the coloring algorithm.

Benchmark	#	#	total size		runtime (ms)	
	nodes	edges	color	perm	color	perm
myciel3	11	20	89792	74688 (16%)	0	0
myciel4	23	71	125120	83520 (33%)	0	10
myciel5	47	236	128064	92736 (27%)	0	40
anna	138	986	297600	156544 (47%)	10	200
david	87	812	296768	201408 (32%)	0	130
le450_15a	450	8168	543296	383296 (29%)	150	14550
le450_5a	450	5714	364160	258112 (29%)	90	3990
le450_5d	450	9757	422080	301568 (28%)	170	16890
queen10_10	100	2940	415744	290560 (30%)	20	1130
queen14_14	196	8372	635456	424640 (33%)	70	13240
queen6_6	36	580	242752	167104 (31%)	0	180
miles500	128	2340	442240	265408 (39%)	20	2230
mulsol	197	3925	940864	686592 (27%)	100	16940
mulsol	184	3916	600384	449216 (25%)	60	5130
inithx	645	13979	761280	481728 (36%)	310	84690
inithx	621	13969	769024	471552 (38%)	290	98280

Table 1: Experimental results.

8 Conclusion

In this paper, we present the importance of memory minimization under the context of systems-on-chip. We then present a new algorithm for the global minimization of memory sizes. The novelty of this technique lies in the observation that memory allocation problem can be efficiently solved if an orientation of the conflict graph is found and such orientation can be fully characterized by a permutation of its vertices, or a permutation of the vertex colors. The algorithm can then be elegantly encoded in the classic iterative improvement framework with a complexity of $O(h(|V| + |E|)$, where h is the number of iterations. This algorithm can quickly converge due to the fact that the size of the solution space is only $\chi!$, where χ is the chromatic number of the conflict graph.

In the future, we will study the interaction of this algorithm with other tasks, such as aggressive inter-procedural dataflow analysis, in the bigger context of memory optimization for system-on-chip.

References

[1] S. S. Bhattacharyya and E. A. Lee. Memory management for dataflow programming of multirate signal processing algorithms. *IEEE Trans. on Signal Processin*, 42(5), May 1994.

[2] F. Catthoor, S. Wuytack, E. D. Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle. *Custom Memory Management Methodology, Exploration of memory organization for embedded multimedia system design*. Kluwer Academic Publisher, Boston, MA, June 1998.

[3] G. Chaitin, M. Auslander, A. Chandra, J. Cocke, M. Hopkins, and P. Markstein. Register allocatinon via coloring. *Computer Languages*, 6(1):47–57, 1981.

[4] P. Grun, F. Balasa, and N. Dutt. Memory size estimation for multimedia applications. In *Workshop on Hardware/Software Codesign*, Seattle, March 1998.

[5] J. V. Meerbergen, P. .Lippens, W. Verhaegh, and A. der Werf. Phedio: high-level synthesis for high throughput applications. *Journal of VLSI Signal Processing*, 9(1/2), January 1995.

[6] H. Murata, K. Fujiyoshi, S. Nakatake, and Y. Kajitani. Vlsi module placement based on rectangle-packing by the sequence-pair. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(12), December 1996.

[7] P. Panda, N. Dutt, and A. Nicolau. *Memory Issues in Embedded Systems-on-chip : Optimization and Exploration*. Kluwer Academic Publisher, Boston, MA, October 1998.

[8] L. Ramachandran, D. Gajski, and V. Chaiyakul. An algorithm for array variable clustering. In *Proceedings of the European Design and Test Conference*, Paris, France, March 1994.

[9] H. Schmit and D. E. Thomas. Synthesis of application-specific memory designs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 5(1), March 1997.

[10] I. Verbauwhede, C. Sheers, and J. Rabaey. Memory estimation for high level synthesis. In *Proceeding of the 31st Design Automation Conference*, San Diego, CA, June 1994.

[11] Y. Zhao and S. Malik. Exact memory size estimation for array computations. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 8(5), October 2000.

[12] J. Zhu. Color permutation: an iterative algorithm for memory packing. Technical Report CE-01-04-01, Electrical and Computer Engineering, University of Toronto, April 2001.

[13] J. Zhu. Static memory allocation by pointer analysis and coloring. In *Design Automation and Test in Europe*, March 2001.