Color Permutation: an Iterative Algorithm for Memory Packing

Jianwen Zhu*

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.

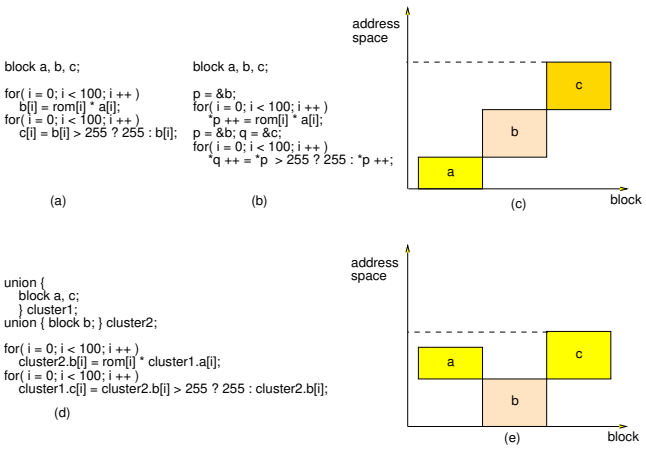


Figure 1: A motivational example.

Consider a motivational example in Figure 1 (a), where memory block `a, b` and `c` needs to be allocated to certain memory space. A naive allocation, as performed by almost all software compilers, is to map each of the block to a distinct memory location, as shown in Figure 1 (c). A careful inspection of the program reveals that block `a` and block `b` can in fact be shared, leading to the allocation in Figure 1 (d), which can be obtained by the modified program in Figure 1 (c).

One might argue that it is the responsibility of the programmer who should identify such opportunities of memory sharing and enforce them the same way as Figure 1 (c) does. We believe that this extra duty is unrealistic for the following reasons. First, the primary goal of a programmer, or a behavior modeler (for the case of hardware synthesis), is to specify functionality, where readability and maintainability

*Jianwen Zhu is with Electrical and Computer Engineering, University of Toronto, 10 King's College Road, Toronto, Ontaria M5S 3G4, Canada.

have higher priority than implementation details. Second, as the application complexity increases, the discovery of memory sharing opportunity becomes intractable to human and automated optimization tools have a better chance to find optimal solution than the programmers.

Simple as it may seem, the memory optimization in Figure 1 is rarely performed in traditional software compilers and behavioral synthesis tools [**?**, **?**]. There are a number of reasons which prevent such optimizations from being incorporated. Among the most fundamental ones is the difficulty of revealing data dependency information for memory blocks under the presence of pointers. For example, Figure 1 (b) performs the same function as Figure 1 (a), except pointer is used to access members of the memory block. While a powerful programming construct, pointer introduces the so-called *memory ambiguity* to the program, which proves to be a killer for data dependency analysis. For example, in Figure 1 (b), it is not clear if `p` is always points to the memory block `b` without sophisticated analysis, hence one has to conservatively assume that the value of `c` may depend on the value of `a`, under which case `a` and `c` can no longer be shared.

While one can alleviate the problem by the use of domain-specific languages or FORTRAN-like array-based languages, where strong assumptions can be made on memory access, the reality is that most system designers use C and its derivatives for system modeling and validation, and they usually exploit the power of pointer constructs for the design of complex data structures and algorithms. While the trend is to directly synthesize C instead of behavioral HDLs into custom hardware as needed [**?**], this paradigm shift is not as simple as a change of synthesis frontend. Among the many challenges is the development of optimization strategies under the presence of pointer constructs. [**?**] has attempted to address this issue in order to apply static memory allocation to general purpose C program by the use of sophisticated pointer analysis techniques.

The data analysis techniques, be it array-based or pointer-based, 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 **??**, we formally define the problem. In Section **??** we present our algorithm in detail. In Section 3, 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 [**?**]. 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 [**?**] 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, [**?**] and [**?**] have studied clustering array variables into different memory blocks. [**?**], [**?**] and [**?**] studied the same problem with the goal of estimation in the context of system level exploration. Philip's Phideo project [**?**], pioneered memory architecture exploration for stream-based signal processing applications. The architecture group at UC, Irvine [**?**] 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 [**?**]. 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

In the text that follows, we use the *formal algorithm notation* (FAN) to state definitions and describe algorithms. Unlike pseudo-code based algorithm description, FAN relies on a type system, where each type is represented by a set, to

present the algorithm in a formal, precise manner. Readers are expected to find this notation very similar to any strongly-typed programming languages and hence straightforward to be translated into implementation, yet abstract enough to allow concise presentation.

The input of the memory allocation problem is a set of memory blocks, as defined in Definition **??**, 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.

Definition 1 *A* **memory block** $v : \mathrm{B}lock$ *is a member of*

```
Block = tuple {                                  1
  size      : N;                                 2
  }                                              3
```

An allocation, as defined by Definition **??** 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 2 *Given a set of memory blocks*[1] $V : \langle\rangle^{Block}$*, 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* $\langle u, v\rangle \in E \rightarrow [\mathcal{A}(u), \mathcal{A}(u) + u.size] \cap [\mathcal{A}(v), \mathcal{A}(v) + v.size] = \emptyset$.

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 **??**, the allocation that results in the smallest total memory size is the optimal allocation.

Definition 3 *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 desribe our proposed allocation algorithm in detail. To offer more insight on why we can perform better, we start by describing the use of graph coloring for memory allocation.

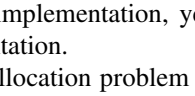
4.1 Graph coloring

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.

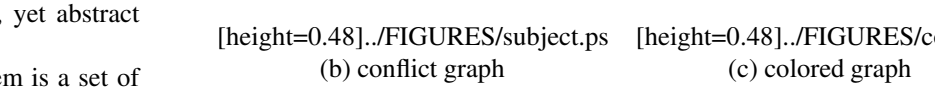
[1] Here we use the notation $\langle\rangle^A$ to represent a power set of A, and the notation $[\,]^A$ to represent the set of all sequences over elements of A.

	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

	a	b	c	d	e	f	g	total
addr	0	1	3	4	1	3	4	7

(d) allocation result

Figure 2: Memory packing by coloring.

Example 1 *Figure* **??** *(a) and (b) shows a conflict graph as well as the sizes of the blocks represented by the vertices of the graph. Figure* **??** *(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.*

Algorithm 1

```
color = func( V : <>^Block, E : V × V ) : V ↦ N {          4
  var σ : [ ]^B;                                           5
  var clr : V ↦ N;                                         6
  var V' : <>^V;                                           7
  var E' : <>^E;                                           8
                                                           9
  V' = V;                                                  10
  E' = E;                                                  11
  while( ||V'|| > 0 ) {                                    12
    v = vertexElimScheme(V', E');                          13
    V' = V' − {v};                                         14
    E' = E' − adjacency(v, E');                            15
    σ = σ ∪ {v};                                           16
    }                                                      17
  forall( v ∈ reverse(σ) ) {                               18
    V' = V' ∪ {v};                                         19
    E' = E' ∪ {v|∃u ∈ V', ⟨u, v⟩ ∈ E};                     20
    clr(v) = min_{∀c∈N, ∀u∈adj(v,E').c≠clr(u)} c;          21
    }                                                      22
  return clr ;                                             23
  }                                                        24
                                                           25
```

The coloring algorithm, as shown in Algorithm **??**, develops a so-called *vertex elimination scheme* σ, a sequence of vertices in V. The reverse of σ is used as the order of assigning colors to vertices. To assign a color to a vertex, one has to search for a color unused by its colored neighbors (Line **??**–**??**).

The choice of the vertex elimination scheme determines the quality and speed of the coloring algorithm. A popular heuristic is to eliminate the vertex with the minimum degree in the current graph.

Algorithm **??** assigns addresses to memory blocks according to the result of coloring. It starts by finding the space required for each color, which should be the maximum size of all memory blocks that are assigned with the corresponding color. It then assigns addresses for each of the colors, which now represent a grouping of memory blocks, by lining them up one by one. The memory address of each block is then found by the address of the corresponding color. It is trivial to show that this allocation algorithm based on coloring has a complexity of $O(|V| + |E|)$.

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 **??** (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 [**?**], 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.

Algorithm 2

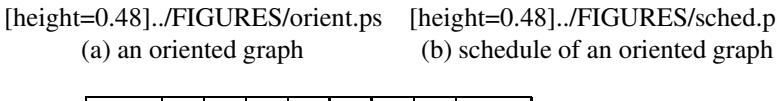
```
allocByColor = func( V : ⟨⟩^Block, E : V × V ) : V ↦ N {
  var clr : V ↦ N;
  var offset : N ↦ N;
  var a : V ↦ N;
  var total : N;

  clr = color(V, E);
  total = max_{v∈V} clr(v);
  forall( c ∈ [0..total] )
    offset(c) = max_{clr(v)=c} v.size;
  forall( c ∈ [0..total − 1] )
    offset(c + 1) = offset(c) + offset(c + 1);
  forall( v ∈ V )
    a(v) = offset(clr(v));
  return a ;
  }
```

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. Definition **??**, Definition **??** and Theorem **??** precisely state that.



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

	a	b	c	d	e	f	g	total
addr	0	1	3	4	1	3	4	6

(c) allocation result

Figure 3: Memory allocation by acyclic orientation.

Definition 4 *Given a conflict graph* $\langle V, E\rangle \subseteq \mathrm{B}lock \times (\mathrm{B}lock \times \mathrm{B}lock)$*, its* acyclic orientation F *is a subset of* E *such that*

- $F \cup F^{-1} = E$ *and* $F \cap F^{-1} = \oslash$*, where* $F^{-1} = \{\langle u, v\rangle | \langle v, u\rangle \in F\}$*;*
- $\nexists [v_0, v_1, ..., v_n] : [\,]^V$*, such that* $\forall i, \langle v_i, v_{i+1}\rangle \in F$ *and* $v_0 = v_n$*.*

Definition 5 *Given a conflict graph* $\langle V, E\rangle \subseteq \mathrm{B}lock \times (\mathrm{B}lock \times \mathrm{B}lock)$*, a schedule of its acyclic orientation* F *is a mapping* $\mathcal{S} : V \mapsto \mathcal{N}$ *such that* $u \prec v \rightarrow \mathcal{S}(u) + u.size \leq \mathcal{S}(v)$*. Here* $\prec$ *is the partial order induced by* F *(or its transitive closure).*

Theorem 1 *Any schedule* $\mathcal{S}$ *for an acyclic orientation* F *of a conflict graph* $\langle V, E\rangle \subseteq \mathrm{B}lock \times (\mathrm{B}lock \times \mathrm{B}lock)$ *is a valid allocation.*

Example 2 *Figure* **??** *(a) shows an orientation of the undirected conflict graph in Figure* **??** *(b). This directed graph can be "scheduled" as shown in Figure* **??** *(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* **??** *(c).*

One can apply any scheduling algorithms to obtained a valid memory allocation. Theorem **??** states that the Algorithm **??**, which employs an ASAP strategy, is in fact optimal for a given orientation.

Algorithm 3

```
asapSchedule = func( V : ⟨⟩^Block, F : V × V ) : V ↦ N {                43
  var ready : ⟨⟩^V;                                                     44
  var count : V ↦ N;                                                    45
  var sched : V ↦ N;                                                    46
                                                                        47
  ready = {v|pred(v) = ∅};                                              48
  while( ready ≠ ∅ ) {                                                  49
    v = choose(ready);                                                  50
    ready = ready − {v};                                                51
    forall( w ∈ succ(v) ) {                                             52
      count(w) = count(w) + 1;                                          53
      if( sched(v) + v.size > sched(w) )                                54
        sched(w) = sched(v) + v.size;                                   55
      if( count(w) = |pred(w)| )                                        56
        ready = ready ∪ {w};                                            57
      }                                                                 58
    }                                                                   59
  return sched ;                                                        60
  }                                                                     61
                                                                        62
```

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. Let F be an acyclic orientation. Then for any schedule S of F, $||S|| \geq ||\mathrm{asap}Schedule(V, F)||$.*

5 Vertex Permutation

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

Definition 6 *A* permutation *of finite set A is a function $P : A \mapsto \mathcal{N}$ such that $\forall u, v \in A.u \neq v \Rightarrow P(u) \neq P(v)$.*

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 vertex permutation $P : V \mapsto \mathcal{N}$, there exists an acyclic orientation F of g.*

Proof: Let $F = \{\langle u, v\rangle \in E | P(u) < P(v)\}$. It follows that $F \cup F^{-1} = E \wedge F \cap F^{-1} = \emptyset$, hence F is an orientation of g. Suppose there exists a cycle $[v_0, v_1, ..., v_k, v_0]$ in F, it follows that $P(v_0) < P(v_1) < .... < P(v_0)$, a contradiction. Hence F is acyclic. □

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 **??**, 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 **??** 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 (Algorithm **??**)can be used to evaluate the solution. Theorem **??** and Corollary **??** 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 [**?**].

[width=0.6]../FIGURES/goodbad.ps

Figure 4: Good and bad orientations.

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, 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 5 *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.*

Proof: Let $F = \{\langle u, v\rangle \in E | C(u) < C(v)\}$. Suppose $\exists \langle u, v\rangle \in E, \langle u, v\rangle \notin F \wedge \langle v, u\rangle \notin F$, then $C(u) = C(v)$, which implies that $\langle u, v\rangle \notin E$, a contradiction. Therefore, F is an orientation. It is trivial to prove that F is also acyclic. □

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 $n!$. Algorithm **??** shows the detail of a greedy search algorithm.

Algorithm 4

```
allocByPerm = func( V : ⟨⟩^Block, E : V × V ) : V ↦ N {                63
  var clr, newClr : V ↦ N;                                             64
  var F : V × V;                                                       65
  var cost, newCost, count : N;                                        66
  var sched, newSched : V ↦ N;                                         67
                                                                       68
  cost = ∞;                                                            69
  newClr = color(V, E);                                                70
  do {                                                                 71
    F = orient(V, E, newClr);                                          72
    newSched = asapSchedule(V, F);                                     73
    newCost = ||newSched||;                                            74
    if( newCost < cost ) {                                             75
      cost = newCost; clr = newClr;                                    76
      sched = newSched; count = 0;                                     77
      }                                                                78
    else                                                               79
      count = count + 1;                                               80
    newClr = perturb(V, clr);                                          81
    } while( count < threshold ) ;                                     82
  return sched ;                                                       83
  }                                                                    84
                                                                       85
orient = func( V : ⟨⟩^Block, E : V × V, clr : V × N ) : V × V {        86
  var F : V × V;                                                       87
  forall( ⟨u, v⟩ ∈ E ) {                                               88
    if( clr(u) < clr(v) )                                              89
      F = F ∪ ⟨u, v⟩;                                                  90
    else                                                               91
      F = F ∪ ⟨v, u⟩;                                                  92
    }                                                                  93
  return F ;                                                           94
  }                                                                    95
                                                                       96
perturb = func( V : ⟨⟩^Block, clr : V ↦ N ) : V ↦ N {                  97
  var c1, c2 : N;                                                      98
  c1 = random(0, max_{v∈V} clr(v));                                    99
  c2 = random(0, max_{v∈V} clr(v));                                    100
  forall( v ∈ V ) {                                                    101
    if( clr(v) = c1 ) clr(v) = c2;                                     102
    else if( clr(v) = c2 ) clr(v) = c1;                                103
    }                                                                  104
  return clr ;                                                         105
  }                                                                    106
                                                                       107
```

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

7.1 Benchmark Methodology

Benchmarking the memory allocation algorithms is not a straight-forward issue for the following reasons:

- The research in this area is still at an early age and hence unlike well-established areas such as logic synthesis, there is no standard benchmarks available.
- Previous work has assumed different computational models, not to mention the different syntax, of the input programs to be optimized, which makes quantitative evaluation of different approaches very hard, since the experiments are difficult to repeat.

[height=0.48angle=270]epic.ps
EPIC
[height=0.48angle=270]jpeg.ps
JPEG
[height=0.48angle=270]mpegdec.ps
MPEGDEC
[height=0.48angle=270]mpegenc.
MPEGENC

Figure 5: Memory block size probability density function

- The memory allocation problem is best solved by breaking down into several smaller problems, for example, dataflow analysis, intra-block allocation and inter-block allocation. The quality of the algorithms for each of the problems is not immediately evident if only the net result is shown.

Since this work focuses on inter-block allocation algorithms, it would be inappropriate to demonstrate the effectiveness of this effort by simply displaying the memory allocation result for an arbitrarily chosen set of C benchmarks, since the accuracy of dataflow analysis algorithms (how to derive the conflict graph), and effectiveness of inter-block algorithms, also play an extremely important role in the final result. Furthermore, these C benchmarks often cannot "stress" the algorithm very well since the problem size that they present is too small to demonstrate the differences between algorithms in terms of space and runtime etc.

Given these considerations, we followed a different experiment methodology to evaluate our algorithm. We obtain the memory conflict graphs directly from the standard DIAMCS benchmark set [**?**] for evaluating coloring algorithms. This solves the comparability problem since it is a standard and available to everyone. In addition, the size of the graph in the benchmark tends to be much larger than the size of graphs in memory allocation problem. Our own experience also shows that the memory conflict graph obtained in real life examples, often exhibit the same characteristics (appearance) as the coloring graph contained in the DIAMCS benchmark.

The DIAMCS graph, on the other hand, does not contain the memory block size information. We opt to generate it randomly. Generating sizes with uniform distribution would be inappropriate since it does not reflect close enough to reality. Instead, we generate memory sizes according to a probability density function (PDF), which is in turn obtained by profiling the real life examples. Figure **??** shows the normalized PDFs for EPIC (Efficient Pyramid Image Coder), JPEG (still image codec), MPEGDEC (MPEG2 decoder) and MPEGENC (MPEG encoder), each of which is taken from the MediaBench benchmark set [**?**].

7.2 Results

We implemented the discussed algorithms in the C programming language and applied them on the DIAMCS benchmarks with randomly generated memory sizes, as described in Section 3.1. The result is summarized in Table **??**: For each benchmark, we show its size in terms of the number

Benchmark	# nodes	# edges	total size		runtime (ms)	
			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
myciel6	95	755	190400	124736 (34%)	10	200
myciel7	191	2360	238080	170496 (28%)	30	880
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_15d	450	16750	792512	598528 (24%)	410	60140
le450_25c	450	17343	919808	653440 (28%)	440	240940
le450_5b	450	5734	352320	245440 (30%)	70	5060
le450_15b	450	8169	550848	385664 (29%)	160	9820
le450_25a	450	8260	756864	540224 (28%)	170	21410
le450_25d	450	17425	901376	653440 (27%)	440	94770
le450_5c	450	9803	391680	292672 (25%)	150	7320
le450_15c	450	16680	789824	573568 (27%)	400	86800
le450_25b	450	8263	767040	505152 (34%)	170	23040
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
queen9_9	81	2112	403648	244032 (39%)	20	960
queen11_11	121	3960	502272	343360 (31%)	30	1470
queen15_15	225	10360	698624	468928 (32%)	90	8700
queen7_7	49	952	318976	215040 (32%)	10	380
queen12_12	144	5192	517440	352000 (31%)	30	6630
queen16_16	256	12640	763776	505216 (33%)	110	19330
queen8_12	96	2736	435840	280896 (35%)	10	1320
queen13_13	169	6656	592768	385984 (34%)	50	5010
queen5_5	25	320	177600	135488 (23%)	10	20
queen8_8	64	1456	361152	236480 (34%)	10	670
miles1000	128	6432	870464	625792 (28%)	60	16500
miles250	128	774	213440	132608 (37%)	10	90
miles750	128	4226	643328	441920 (31%)	30	5200
miles1500	128	10396	1344448	1080320 (19%)	170	58790
miles500	128	2340	442240	265408 (39%)	20	2230
mulsol	197	3925	940864	686592 (27%)	100	16940
mulsol	184	3916	600384	449216 (25%)	60	5130
mulsol	186	3973	642240	453760 (29%)	70	8740
mulsol	188	3885	609920	463296 (24%)	60	6450
mulsol	185	3946	613376	451584 (26%)	50	9510
inithx	864	18707	1143552	787776 (31%)	540	268780
inithx	645	13979	761280	481728 (36%)	310	84690
inithx	621	13969	769024	471552 (38%)	290	98280

Table 1: Experimental results.

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. Although not shown in Table **??**, we found that the proposed algorithm performs significantly better, although with longer runtime, than the algorithm we proposed in an earlier unpublished study [**?**], which used heuristic acyclic orientation to obtain an allocation, but without further iterative improvement. We also found that algorithm can achieve better results with less runtime than the cubical algorithm described in [**?**].

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.