Static Memory Allocation by Pointer Analysis and Coloring

Jianwen Zhu
Electrical and Computer Engineering
University of Toronto, Ontario M5S 3G4, Canada

Abstract

Modern systems-on-chips often allocate more silicon real-estate on memory than logic. The minimization of on-chip memory becomes increasingly important for the reduction of manufacturing cost. In this paper, we present a new technique that minimizes memory usage. Incoporated in a behavioral synthesis tool that synthesizes general-purpose C programs, this technique is fully automated and does not rely on users to explicitly specify dataflow information. Experimental results show that significant improvements can be achieved for the benchmark set.

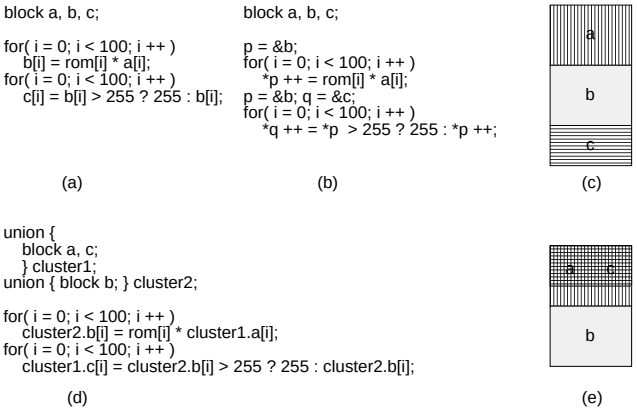

Figure 1. A motivational example.

1 Introduction

Today's telecommunication and consumer electronics applications demand computational power that can be met only by integrating more and more hardware components on a single chip. 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 research has devoted to the minimization of logic area in order to reduce manufacturing cost, which exponentially depends on the die size, the minimization of memory size of general purpose programs has largely been ignored.

Consider a motivational example in Figure 1 (a), where memory blocks `a, b` and `c` need to be allocated to certain memory area. A naive allocation, as performed by almost all the software compilers, is to map each of the block to distinct memory locations, 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 (e), which can be obtained by the modified program in Figure 1 (d).

One might argue that the programmer should identify such opportunities of memory sharing and enforce them the same way as Figure 1 (c) does. We believe this requirement 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, and readability and maintainability has higher priority than implementation details. Second, as the application complexity increases, 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 [5, 8]. 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 is known to be the killer for data dependency analysis. For example, in Figure 1 (b), it is hard to conclude that `p` always points to the memory block `b` without sophisticated

analysis. As a result, 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 opt to use domain-specific languages to alleviate the problem, 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 [9], this paradigm shift is not as simple as a change of language frontend. Among the many challenges is the development of optimization strategies under the presence of pointer constructs.

In this paper, we focus on the problem of static memory allocation for general purpose C programs. We achieve our goal of memory minimization by discovering chances of memory sharing. The major contribution of this paper is that unlike traditional works which either depend on explicit dataflow information, or regular array data structure, we rely on the use of sophisticated pointer analysis techniques to reveal the memory access patterns with reasonable accuracy, and hence we are able to extend the scope of programs that the memory sharing optimization can apply.

The rest of the paper is organized as follows: In Section 2, we discuss the related works. In Section 3, we make a number of simplifying assumptions and state the problem formulation. In Section 4 we present our algorithms in detail. In Section 5, we show the experimental results.

2 Related Work

The memory placement problem for instruction and data cache performance optimization has been studied extensively in the compiler community. More recently, Panda et. al. [10] have studied this subject for embedded software and hardware.

The memory size minimization problem has been considered under two occasions: First, 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 and hence the memory minimization problem is well formulated. Second, a number of research groups, for example, the IMEC Acropolis group [2], and Irvine ACES group [6], have studied the memory minimization problem for regular data structure. In their study, efficient analysis techniques are used to optimize programs, typically those in multimedia domain, which operate on multi-dimensional arrays.

The major difference between our work and the abovementioned efforts is that we target pointer-intensive applications, and we focus only on sequential programs. The practical implication of this is that our tool can handle arbitrary C codes with arbitrary complex data structures. Given the fact that most applications are first developed and validated in C, and the amount of effort involved in transforming them into dataflow programs, our tool can be attractive for those who would like to directly synthesize C programs into hardware.

Our tool relies on pointer analysis technique, which has been an intensive research area in its own right. Pointer analysis techniques can be classified as context-insensitive [12], and context-sensitive [14], with the latter having higher accuracy but also higher complexity. The pointer analysis techniques are traditionally used for instruction-level performance improvement. Recently, [7] uses reference analysis (the simplified, Java version of pointer analysis) for coarse-grained parallelism. [11] uses the analyzer of [14] for pointer resolution in hardware. Compared to these efforts, our contribution is the new application of pointer analysis technique for memory minimization.

When enough information is collected by analysis, we use graph coloring algorithm for memory sharing. The same algorithm has been used for the problem of register allocation [3].

3 Problem Formulation

This paper tackles the static memory allocation problem, which assigns addresses to statically declared variables in a program in such a way that the total memory usage is minimized without alternating the program behavior.

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 way. For example, the type $\langle\rangle^A$ represents a power set of A, and the type $[\,]^A$ represents a sequence of elements in A. 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.

We make several simplifying assumptions for the sake of *concise presentation*.

Assumption 1 *The program does not contain dynamic memory allocation.*

According to Assumption 1, there should be no system calls `malloc` and `free` involved in a C program. Note that relaxing this assumption does not break our tool since it is not our tool's job to manage dynamic memory anyway. One can fold all dynamically allocated memory into one single memory block and apply our proposed algorithm easily.

Given that, we can safely assume that all memory operations are performed on a set of memory blocks with known size at compile time. This memory block set is formalized in Definition 1.

Definition 1 *A* **memory block** b : B*lock is a member of*

B*lock* = **tuple** { 1
  size : $\mathcal{N}$; 2
  } 3

Since the statically declared variables can be potentially accessed by all procedures, one needs an inter-procedural analysis framework in order to carry out the optimization. We assume that such a framework exists and it effectively provides the illusion that all procedure calls are inlined and the entire program can be represented as a single control flow graph.

Assumption 2 *A behavioral program can be represented as a control flow graph, as defined in Definition 2.*

Definition 2 *A* **control flow graph** g : Cfg *a member of*

Cfg = **tuple** { 4
  V : $[\,]^{Instrn}$; 5
E : $V \times V$; 6
  } 7

Here, the vertices of the control flow graph represents a basic block, or a sequence of non-branching instructions, as defined in Definition 3, and edges capture the control flow information.

Assumption 3 *The instructions in the control flow graph are in static single assignment form (SSA) [4].*

Definition 3 *An* **instruction** i : I*nstrn is a member of*

I*nstrn* = **tuple** { 8
  opcode : $\{LD, ST, AC, IC, FC, OP, PHI\}$; 9
  srcs : $[\,]^{Instrn \cup Block \cup \mathcal{N} \cup \mathcal{R}}$; 10
  } 11

Each instruction is characterized by its opcode and operands. The LD opcode indicates a memory load operation, whose only operand indicates the address of the memory location. The ST opcode indicates a memory store instruction, whose operands indicate the address and value respectively. The AC opcode indicates an address constant instruction, which takes a memory block as its operand. The IC opcode indicates an integer constant instruction, which takes an integer number as its operand. The FC opcode indicates a floating-point constant instruction, which takes an floating point number as its operand. The PHI instruction is an artifact of SSA analysis [4], which combines all its operands, each of which coming from a different branch, into a single value. The OP opcode indicates the set of instructions which do not have side effects on memory, such as addition and subtraction. The differences between these instructions are not made since they are irrelevant for the subsequent analysis.

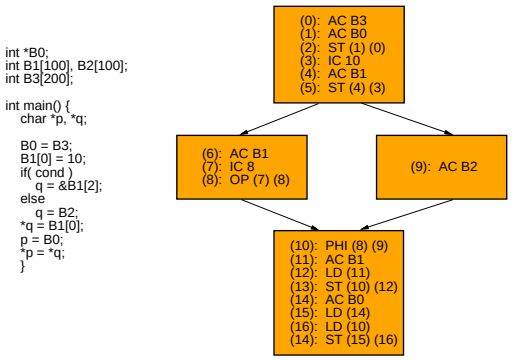

Figure 2. An example C program and its control flow graph.

Under these assumptions, we can formulate the static memory allocation problem as follows.

Definition 4 *Given a static memory set* B : $\langle\rangle$*Block and a control flow graph* g : Cfg*, a* **static memory allocation** *is a function* $\mathcal{A} : B \mapsto \mathcal{N}$ *such that*

- *the total memory size* $\Sigma_i max_{b \in B.\mathcal{A}(b)=i} size_b$ *is minimized;*
- $\forall b_1, b_2 \in B, \mathcal{A}(b_1) = \mathcal{A}(b_2) \Rightarrow$ *for every execution trace of g, lifetime of* b_1 *does not overlap with that of* b_2*. Here the lifetime of a memory block for an execution trace is defined by the time between it is first written and last read.*

4 Static Memory Allocation Algorithm

Our proposed algorithm proceeds with four steps.

In the first step, control flow analysis is performed to obtain the *dominance tree D* (Line 19). A control flow graph node *a* is said to dominate, or be the dominator of, another node *b* if every execution path which passes *b* also passes *a*. A dominance tree is established by adding an edge to a node from its immediate dominator. The detailed algorithm

for deriving the dominance tree is ignored here since it is a well-known algorithm.

In the second step (Line 20), pointer analysis is performed to obtain the *point-to function*, which maps each instruction contained in the control flow graph to the set of possible memory blocks that the result value of the instruction may point to.

In the third step (Line 21), given the dominance tree and point-to function, *liveness analysis* is performed to constructs the *interference graph*, where each vertex in the graph represents a memory block, and each edge between two vertices indicates that the corresponding blocks cannot be shared due to the overlap of their life times.

Finally (Line 22), the interference graph is colored to obtain the allocation result.

Algorithm 1

```
memAlloc = func(                                          12
    B : ⟨⟩^Block, g : Cfg                                 13
    ) : B ↦ 𝒩 {                                           14
    var D : V_g × V_g;                                    15
    var p : Instrn ↦ ⟨⟩^B;                                 16
    var I : B × B;                                        17
                                                          18
    D = dominanceAnalysis(B, g);                          19
    p = pointerAnalysis(B, g);                            20
    I = liveAnalysis(B, g, D, p);                         21
    return coloring(B, I) ;                               22
    }                                                     23
                                                          24
```

4.1 Pointer Analysis

As stated earlier, the aim of pointer analysis is to give an estimate of the runtime values of all instructions. Of particular interests to the pointer analysis are those runtime values which happen to be the addresses of *memory locations* within the memory blocks under study, called the *pointer values*. The challenge of pointer analysis is that all imperative programs maintain a runtime state, which itself consists of memory blocks. Since the program state may contain pointer values, and is constantly retrieved and updated by the LD and ST instructions, it is mandatory that the program state is kept track of. In other words, the possible address values that each memory location may assume must be maintained at every program point. Unfortunately, unlike instructions themselves, the number of memory locations is unbounded since the size of program state is unlimited (if dynamic memory is involved) and every memory word can potentially hold an address value. Therefore, approximations need to be introduced to reduce the computational cost.

Assumption 4 *Different elements of the same array are not distinguished.*

According to Assumption 4, all array elements are folded into a single location set and pointer arithmetic does not change the corresponding pointer value. Note that this assumption is made by all the pointer analysis tools due to the fact that array indexes can not be inferred at compile time, with the exception of loop induction variables.

Assumption 5 *Different fields of the same record are not distinguished.*

According to Assumption 5, addresses with constant offsets from a memory block are also folded into a single location set. Note that this assumption is presented here only for the simplification of presentation since together with Assumption 4, there is no need to distinguish different locations in the same memory block, and we can simply use memory blocks as point-to values. For a detailed treatment where the distinction is made, the readers are referred to [14].

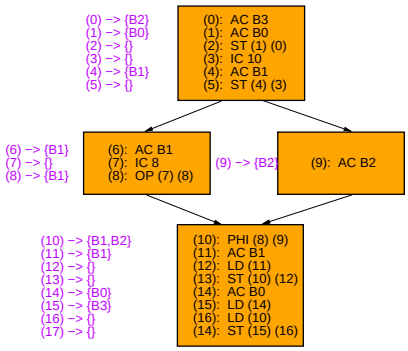

Figure 3. Pointer analysis.

Algorithm 2 shows the simplified pointer analysis algorithm. Note that p maintains the pointer values of all instructions, $state$ maintains the pointer values of program state at the end of each basic block, and $state$ maintains the pointer values of program state at each program point.

The algorithms is formulated under a standard dataflow analysis framework. For each iteration, it traverses the control flow graph in preorder. For each control flow graph node v, it first combines all the program state pointer values of all its predecessors into $state(v)$. It then computes the pointer value for each instruction. When an LD instruction is encountered, the pointer value is retrieved from $state(v)$. Likewise, when a ST instruction is encountered, $state(v)$ is updated accordingly. The program terminates until the fixed-point condition is reached, that is, the point-to value of every instruction is not changed.

Algorithm 2

```
pointerAnalysis = func(                                          25
   B : ⟨⟩^Block, g : Cfg                                         26
   ) : Instrn ↦ ⟨⟩^B {                                           27
   var p : Instrn ↦ ⟨⟩^B;                          ;             28
   var state : V_g ↦ (B ↦ ⟨⟩^B);                                 29
   var changed : {true, false};                                  30
   var new : ⟨⟩^B;                                               31
                                                                 32
   while( ( ) changed ) {                                        33
   changed = false;                                              34
                                                                 35
      forall( v ∈ preorder(V_g, E_g) ) {                         36
         forall( b ∈ B )                                         37
            state(v)(b) = ∪_{u∈pred(v,E_g)} state(u)(b);          38
         forall( i ∈ v ) {                                       39
            if( opcode_i = IC ∨ opcode_i = FC )                  40
               new = ∅;                                          41
            else if( opcode_i = AC )                             42
               new = {src_i[0]};                                 43
            else if( opcode_i = OP )                             44
               new = p(src_i[0]);                                45
            else if( opcode_i = PHI )                            46
               new = ∪_{s∈src_i} p(s);                           47
            else if( opcode_i = LD )                             48
               new = ∪_{b∈p(src_i[0])} state(b);                 49
            else if( opcode_i = ST ) {                           50
               forall( b ∈ p(src_i[0]) )                         51
                  state(i)(b) ∪= p(src_i[1]);                    52
               }                                                 53
            if( new ≠ p(i) ) {                                   54
               p(i) = new;                                       55
               changed = true;                                   56
               }                                                 57
            }                                                    58
         }                                                       59
      }                                                          60
   return p ;                                                    61
   }                                                             62
                                                                 63
```

Figure 3 shows how the example in Figure 2 is annotated with the point-to information at the left hand side.

4.2 Inference Graph Construction

Next, the interference relation between memory blocks has to be established.

To achieve this, one has to first find out where the *definition point* of each memory block, that is, where it is first assigned a value. Note that the answer may vary for different control flow graph nodes if they happen to belong to the different branches of the program. On the other hand, the answer should be the same for all the nodes along a particular path of the control flow graph. The algorithm tries to construct the def, which maps a memory block to its defining instruction for each flow graph node, by traversing the flow graph in the preorder of its dominance tree D. For each node traversed, it first inherits the definition point information from its immediate dominator. Then for each ST instructions it encounters, it updates the definition point information if it is the first time that the corresponding memory block is assigned.

The algorithm then traverses the control flow graph backwards by computing the *live set* of each node, that is, the set of memory blocks that are alive at the beginning of the node. The live set of each node is initialized as the union of the live sets of all its successors. Given the order of traversal, we are assured that the live sets of successors have been computed. Instructions in the node is then examined in the reverse order: First, whenever an LD instruction is encountered, the corresponding memory blocks are added to the live set. Second, whenever a definition point is encountered, the corresponding memory blocks are excluded from the live set. Third, whenever a memory access instruction is encountered, interference edges between the corresponding memory blocks and the elements of the live set are constructed.

Finally, the interference edge set is returned.

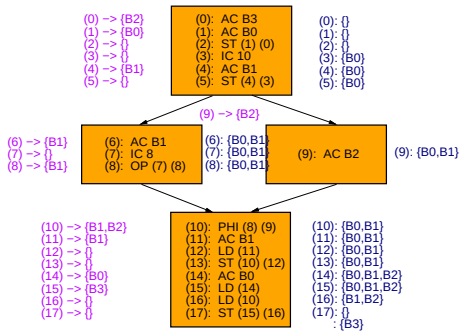

Figure 4. Liveness analysis.

Figure 3 shows the example in Figure 2 annotated with the live set information for each instruction at the right hand side of each basic block.

4.3 Graph Coloring

Given the interference graph, graph coloring can be performed to obtain a clustering of memory blocks. That is, the memories that are assigned the same color are to be assigned the same memory address. The algorithm employs the heuristics used in [3].

5 Experimental Result

We performed experiments on the Toronto DSP benchmark suite developed and distributed by Lee and Chow [13]. This benchmark suite contains both the kernel benchmarks, which consist of small programs extracted from DSP applications, and the application benchmarks, which consist of complete programs performing particular tasks. Since our tool is targeted towards exploiting memory sharing for large programs, it makes more sense that we examine application

Benchmark	#lines	size w/t opt.	size w/o opt.	saving
adpcm	489	2,336	2,336	0%
spectrum	321	2,032	2,032	0%
compress	324	70,912	70,656	0.3%
edge_detect	224	327,752	196,644	40%
histogram	87	133,120	67,584	50%
lpc	632	7,388	5,708	23%

Table 1. Memory saving for Toronto DSP Benchmark Suite.

benchmarks. While most of these benchmarks use arrays extensively, our experiments are carried out on those where arrays are accessed using pointers and hence array-based analysis becomes insufficient.

Table 1 summarizes the memory saving we can achieve for the following benchmarks: `adpcm`, `spectrum`, and `lpc` are various speech processing applications; and `compress`, `edge_detect`, and `histogram` are image processing applications. The complexity of each benchmark is given in the first column in terms of number of lines of C code. The second column gives the memory usage without optimizations (note that a few benchmarks are modified to remove the manual optimization). The third column gives the memory usage if optimization is performed using our tool. The forth column gives the percentage of memory saving we are able to achieve.

We observe that the more complex the data structure an application has, the more chances it may have for memory minimization. For example, the `adpcm` benchmark has only two arrays for input and output, as well as several arrays to keep filter coefficient, in which case our tool does not really help. On the other hand, for the `lpc` benchmark, where there are twelve arrays, each of which may have different sizes and different logical purposes, we can compress the memory by merging arrays with non-overlapping lifetimes. Note that this is also consistent with the intuition that the more complex the application is, the more unlikely the programmers can afford to perform manual memory optimization.

6 Conclusion

In this paper, we have presented the importance of memory minimization under the context of systems-on-chip. We then presented a new technique for the global minimization of memories. Unlike previous work, this technique can handle arbitrary C programs, including those which use pointers intensively. We demonstrate the effectiveness of this approach by applying the proposed technique on DSP benchmarks. Future work will extend to the study of networking applications, which contains irregular data structure, as well as algorithm improvement, for example, enhancing the accuracy of pointer analysis, as well as the efficiency of memory allocation.

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] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, October 1991.

[5] D. Gajski, N. Dutt, A. Wu, and S. Lin. *High Level Synthesis: Introduction to Chip and System Design.* Kluwer Academic Publishers, 1992.

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

[7] R. Helaihel and K. Olukotun. Java as a specification language for hardware-software systems. In *Proceedings of the International Conference on Computer-Aided Design*, November 1997.

[8] G. D. Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw Hill, 1994.

[9] G. D. Micheli. Hardware synthesis from c/c++ models. In *Proceedings of the Design Automation and Test Conference in Europe*, March 1999.

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

[11] L. Semeria and G. D. Micheli. Spc: Synthesis of pointers in c: Application of pointer analysis to the behavioral synthesis from c. In *Proceedings of the International Conference on Computer-Aided Design*, pages 321–326, November 1998.

[12] B. Steensgaard. Efficient context-sensitive pointer analysis for c programs. In *Proceedings of the 1996 International Conference on Compiler Construction*, pages 136–150, April 1996.

[13] `http://www.eecg.toronto.edu/~corinna`.

[14] R. Wilson and M. Lam. Efficient context-sensitive pointer analysis for c programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, June 1995.