Memory Representation and Hardware Synthesis of C Code with Pointers and Complex Data Structures

Luc Séméria
lucs@azur.stanford.edu
Computer Systems Laboratory
Stanford University

Koichi Sato
koichi@lsi.nec.co.jp
System LSI Design Engineering Division
NEC corporation

Giovanni De Micheli
nanni@galileo.stanford.edu
Computer Systems Laboratory
Stanford University

Abstract -- *We present our tool SpC which enables the synthesis of C behavioral models with pointers and complex data structures. For both analysis and synthesis, memory is represented by* location sets. *During memory partitioning, these location sets are either mapped to simple variables or arrays. Pointers are encoded and loads/stores are replaced by assignments in which data are directly accesses. Finally, dynamic memory allocation and deallocation are performed within user-defined* memory segments *by an optimized hardware allocator instantiated from a library.*

1. INTRODUCTION: SYNTHESIS FROM C

Different languages have been used as input to high-level synthesis. Hardware Description Languages (HDLs), such as Verilog HDL and VHDL, are the most commonly used. However, designers often write system-level models using programing languages, such as C or C++, to estimate the system performance and verify the functional correctness of the design. Using C/C++ offers higher-level of abstraction, fast simulation as well as the possibility of leveraging a vast amount of legacy code and libraries, which facilitate the task of system modeling.

The use of C/C++ or a subset of C/C++ to describe both hardware and software would accelerate the design process and facilitate the software/hardware migration. Designers could describe their system using C. The system would then be partitioned into software and hardware blocks, implemented using compilers and synthesis tools.

In order to help designers refine their code from a simulation model to a synthesizable behavioral description, we are trying to efficiently synthesize the full ANSI C standard. This task turns out to be particularly difficult because of dynamic memory allocation, function calls, recursions, `goto`'s, type castings and pointers.

Different subsets of C and C-like HDLs have been defined and used for synthesis. We mention first those developed in the eighties. HARDWAREC [4] is a fully synthesizable language with a C-like syntax and a cycle-based semantic. It doesn't support pointers, recursion and dynamic memory allocation. CONES [12] from AT&T Bell Laboratories is an automated synthesis system that takes behavioral models written in a C-based language and produces gate-level implementations. Here, the C model describes circuit behavior during each clock cycle of sequential logic. This subset is very restricted and doesn't contain unbounded loops nor pointers.

In the recent past a few projects have been looking at means to use C/C++ as an input to current design flow [6]. The general idea is to both extend and restrict the C/C++ languages. Constructs are added to the languages to model coarse-grain parallelism, communication and data-type. For reactivity, SYSTEMC [21] from Synopsys, CoWare and Frontier Design supports a mixed synchronous and asynchronous approach implemented as a C++ library. Other extensions include ECL [5] from Cadence based on C and Esterel, HANDLE-C [17] and BACH-C [3] originally based on OCCAM, SPECC [18] based on SPECCHART and CYNLIB [16]. In order to map functionality to hardware, a synthesizable-C/C++ subset is usually defined. We can distinguish two approaches. The first approach consists of translating a subset of C into HDL (Verilog or VHDL) which will eventually be synthesized using today's synthesis tools. Examples of such approach include the early BACH-C compiler [3] from Sharp, OCAPI [8] from IMEC as well as other commercial tools. The second approach consists of using C/C++ directly, as an input to behavioral synthesis. In particular, this approach has been chosen by Synopsys with SCENIC [2] and by NEC with CYBER [13].

C/C++ is a procedural imperative language. Its semantic relies on an implicit Von Neuman architecture. The implementation of sequential functional descriptions into hardware has extensively been studied during the last decade. Synthesis from C/C++ descriptions can leverage some of this work but also requires the development of some extensions for efficiently supporting the different constructs of C/C++ such as pointers, complex data structures, dynamic memory allocation, and object oriented features. In particular, the synthesis of C code involving dynamic memory allocation requires the access to an operating system running in software or the generation of hardware allocators such as these implemented in the MATISSE framework [15].

The overall objective of our research is to explore synthesis from full ANSI C. In our tool SpC [9], pointer variables are resolved at compile-time to synthesize C functional models in hardware efficiently. An extension to handle dynamic memory allocation (`malloc/free`) has also been presented [11]. In this paper, we focus on the mapping of complex data structures into hardware. Besides, we present how arrays of pointers as well as pointers inside of structures can be efficiently mapped to hardware. Two examples of implementations are also presented.

2. MEMORY REPRESENTATION

In software, C programs are targeted to a virtual architecture consisting of one memory in which everything is stored. Even though `register` declaration may allow programmers to specify the variables to place in registers, the assignment of variables to registers is generally done by the compiler. The notion of caches and memory pages are transparent to programmers.

In hardware, at the behavioral level, designers want to have control on where data are stored and want to optimize the locality of the storage. Typically, data may be stored in multiple memory banks, registers, and wires (e.g. output of a functional unit). For HDLs, the allocation of storage and the mapping of data to storage can be integrated with high-level synthesis. However, for C models, partitioning the memory is complicated by such constructs as pointers, out-of-bounds array accesses, type casting, and dynamic memory allocation.

In order to efficiently map C code into hardware, one need an accurate representation of the memory. Such information is also widely used in compilers. In order to parallelize programs onto distributed architectures, the independent sets of data which can be processed in parallel have to be extracted.

The simplest memory representation consists of a single address space in which all data are stored. This trivial representation however prevents from optimizing the locality and parallelizing the code. On the other hand, the most accurate representation, that would distinguish each element of arrays or of recursive data structures, is not practical. As a result most analysis techniques combines elements within a single data structure.

In order to find both an accurate and a practical representation for hardware synthesis, we propose to use the notion of *location sets* introduced by Wilson and Lam [14]. Locations sets support any of the data structures available in C including arrays, structures, arrays of structures and structures containing arrays. This representation is also relatively simple as it combines the different elements of an array or of recursive data structures. It can therefore be used for large C programs.

Let B be the set of memory blocks corresponding to the different variable declarations. A *location set* $l = \langle loc, f, s \rangle \in B \times \mathbb{N} \times \mathbb{Z}$ represents the set of locations with offsets $\{f + is | i \in \mathbb{Z}\}$ in a particular block of memory *loc*. That is f is an *offset* within a block and s is the *stride*. If the stride is zero, the location set contains a single element. Otherwise, it is assumed to be an unbounded set of locations. Table 1 shows the location sets for various expressions.

Type	Expression	Location Set
`int a`	`a`	$\langle$a, $0, 0\rangle$
`struct {int F;} s`	`s.F`	$\langle$m, $f, 0\rangle$
`int a[];`	`a[i]`	$\langle$a, $0, s\rangle$
`struct {int F;} r[];`	`r[i].F`	$\langle$r, $f, s\rangle$
`struct {int F[10];} r;`	`r.F[i]`	$\langle$r, f mod$s, s\rangle$

Table 1: Location set examples (f=offset of field F), (s=stride or array element size)

For simple data structures (arrays, structures, array of structures), offsets are used to identify the different fields of structures whereas strides are used to record array-element sizes. Figure 1 gives an example of representation for an array of structures. The representation doesn't distinguish the different elements within the array but it distinguishes the different instantiations of variables and structures. This makes sense since all elements of an array are usually alike.

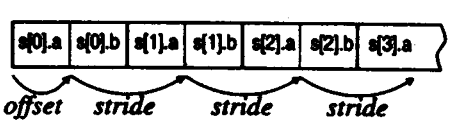

Figure 1: Representation of `struct{int a; int b} s[]`

Nested arrays and structures, type casting and pointer arithmetic are making things more complicated leading to some more inaccuracies. The array bound information in the declared type cannot be used because the C language does not provide array-bounds checking. A reference to an array nested in a structure could access other elements of the structure by using out-of-bound array indices.

Dynamically allocated memory locations (heap-allocated objects) are represented by a specific location set. As far as accuracy, the goal is to distinguish complete data structures. The different elements of a recursive data structure would typically be combined. For example, we want to distinguish one list from another but we do not want to distinguish the different elements of a list. Storage allocated in the same context is assumed to be part of the same equivalence class. This heuristic have been proven to work well as long as the program uses the standard memory allocation routines [14].

In order to generate an accurate memory representation, one also needs to analyze the pointers' usage. Pointer analysis is a compiler technique to identify at compile-time the potential values of the pointers in the program. It determines the set of locations the pointer may point to (point-to information). This information is not only used to create the memory representation but it is also used for synthesis. In the case of *loads* (`...=*p`), *stores* (`*p=...`), and `free`, we want to synthesize the logic to access, modify or deallocate the location referenced by the pointer. The point-to information must be both *safe* and *accurate*: *safe* because we have to consider all of the locations the pointer *may* reference and *accurate* because the smaller the point-to-set is, the more accurate the memory representation is and the less logic we have to generate. We use a flow- and context-sensitive pointer analysis [14] which provides better accuracy compared to other analyses. Even though the complexity of flow- and context-sensitive analyses may be exponential, it is not a limitation for hardware synthesis because we deal with rather small and simple programs with limited calling contexts for functions and often no recursions.

The pointer analysis and memory representation used here support the complete ANSI C syntax. In this paper however, we define our own synthesizable subset. Our subset includes `malloc`/`free` as well as all types of pointers and type casting. Nevertheless we set the following two restrictions.

The first restriction applies to systems described as a set of parallel processes: pointers that reference data outside of the scope of a process (e.g. global variables or data internal to some other processes) are not allowed. The second limitation stems from the fact that most commercial synthesis tools have restrictions on functions. Recursions are usually not supported. Procedures that are mapped to components typically have restrictions both on their functionality and their parameters (e.g. parameter passed by reference is not supported by most HDL syntax). The synthesis of functions in C is beyond the scope of this paper. Functions in general are supposed to be inlined prior to synthesis.

3. MAPPING TO HARDWARE

After analysis of the program, memory can be represented as a set of location sets. Each location may represent a unique location (case of a stride null), multiple locations (stride not null) or heap objects. During mapping to hardware, each location set is mapped to a single variable or an array that can be synthesized using current synthesis tools.

3.1 Partitioning of the memory

The memory is partitioned into a set of locations sets. In this section, we do not consider pointers and heap objects. The synthesis of pointers and `malloc`/`free` is presented in Sections 3.2 and 3.3. In the rest of the paper, we use the following representation for *fundamental* (or basic) types: `char` and `unsigned char` are represented as 8 bits, `short` and `unsigned short` are represented as 16 bits, and `int` and `unsigned int` are represented as 32 bits. These representations are the most common on 32 bits architecture. Derived types such are pointers, arrays and structures are constructed from these fundamental data types.

Without heap objects, we can distinguish two types of location sets: *unique location sets* whose stride are null, and *multiple locations sets* with non-zero stride. For each location set *<loc, f, s>*, we define the variable SPC_*loc_f_s*.

For unique location set (*s* null), SPC_*loc_f_s* is a variable of fundamental type. In the case of a location set representing a variable of fundamental type (e.g. `char`, `short`, `int`) the mapping is straightforward. For structures, their different fields can be mapped to separate variables (akin to registers in the final hardware) as long as they are represented by separate location sets.

For multiple location set (*s* not null), SPC_*loc_f_s* is defined as an array of fundamental type elements (e.g. array of integers). These arrays can then typically be mapped to memories or register files either manually or according to current methodology [1,7]. For arrays of structures, the different fields of the structures can be mapped to different memories as long as their representations do not overlap. This allows to independently access the different fields of the structures, leading to more flexibility and potentially better performances.

Example 1. *Consider the following structure variable.*

```
struct {
    char  c1;
    char  c2;
    short s;
    int   i;
} csi;
```

Four location sets represent the four fields of the structure `csi`. *On our specific target architecture, the fields* `csi.c1`, `csi.c2`, `csi.s`, *and* `csi.i` *are respectively represented by the location sets* <`csi`, *0, 0*>, <`csi`, *1, 0*>, <`csi`, *2, 0*>, *and* <`csi`, *4, 0*>. *The layout in memory before synthesis is represented on Figure 2.*

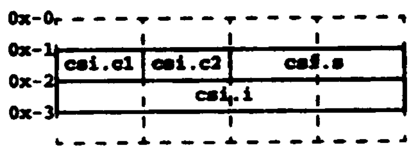

Figure 2: memory layout of `struct {char c1; char c2; short s; int i} csi`.

We create the following variables corresponding to each location set:

```
char SPC_csi_0_0;  // csi.c1
char SPC_csi_1_0;  // csi.c2
short SPC_csi_2_0; // csi.s
int SPC_csi_4_0;   // csi.i
```

As a result during the mapping to hardware the assignment

```
csi.c2 = 0;
```

is replaced by

```
SPC_csi_1_0 = 0;
```

Out of bound array accesses, as well as copies of structures can make things more complicated. With our memory representation, one data (e.g. an entire structure) may be represented by the concatenation of multiple elements of location sets. In Example 2 a structure is represented as two integers. In Example 3, an integer inside of a structure is represented by the concatenation of two short integers.

Example 2. *This example illustrates the implementation of a structure copy.*

```
struct { int x; int y } A, B;
A = B;
```

After translation, the following synthesizable code is generated:

```
int SPC_A_0_0, SPC_B_0_0;   // A.x, B.x
int SPC_A_4_0, SPC_B_4_0;   // A.y, B.y

// A = B;
SPC_A_0_0 = SPC_B_0_0;
SPC_A_4_0 = SPC_B_4_0;
```

The structure copy is broken into two assignments corresponding to the two fields of the structure.

Example 3. *In the following code segment, the structure variable* `its` *contains an array of short integers.*

```
struct {
    int i;
    short ts[2];
} its;
int a, b;

its.i = a;
b = its.i;
```

Because of potential out of bound array accesses (e.g. `its.t[-1]`*), the structure variable* `its` *is entirely represented by the location set* <`its` *, 0, 2*>. *The code segment is then transformed into:*

```
short SPC_its_0_2[4];
int SPC_a_0_0, SPC_b_0_0;

// its.i = a;
SPC_its_0_0[0] = SPC_a_0_0 >> 16;
SPC_its_0_0[1] = SPC_a_0_0;

// b = its.i;
SPC_b_0_0 = SPC_its_0_0[0] << 16
            SPC_its_0_0[1];
```

Note that, using a concatenation operator {...}, *these assignments can be written as:*

```
{
  SPC_my_str_0_0[0]
  SPC_my_str_0_0[1]
} = SPC_a_0_0;

SPC_b_0_0 = {
              SPC_my_str_0_0[0]
              SPC_my_str_0_0[1]
            };
```

3.2 Pointers

In the previous section, we did not consider pointer and type casting. In software, the semantic of pointers is the address of data in memory. This semantic assumes the target architecture consists of a single memory space in which all data are stored.

In hardware, as discussed in Section 2, data may be stored in multiple registers, memories or even wires (e.g. output of a functional block). Therefore, to efficiently map C code into hardware, pointers may not only address data in memory, they may also reference registers, wires or ports. Our synthesis tool generates the appropriate circuit to dynamically access these locations according to the pointers' value.

Pointers can be used to allocate, read, write and deallocate data. Allocation and deallocation performed through the standard library functions `malloc` and `free` are dealt in the next section. For loads (...=*p) and stores (*p=...), we distinguish two types of pointers: pointers to a single location, which can be removed, and pointers to multiple locations.

Loads from pointers to a single location are simply replaced by assignments from the location accessed. Similarly, stores are simply replaced by assignments to the location referenced. During memory partitioning, these locations are mapped to location sets. As seen previously in Examples 2 and 3, location accessed may correspond to the concatenation of multiple location set elements. Moreover, because of pointer type casting, the location on which the load or store is performed may correspond to only part of a location set element, as shown in Example 4.

Example 4. *Consider the following code segment in which we have a load and a store with type casting from type pointer to integer (*`int *`*) to type pointer to short integer (*`short *`*).*

```
short s[2];
int i;

s[0] = *(short *)&i;      16 LSB of i
*(short *)&i = s[1];
```

The code segment is transformed into:

```
short SPC_s_0_2[2];
int SPC_i_0_0;

SPC_s_0_2[0] = SPC_i_0_0>>16;
SPC_i_0_0 = SPC_s_0_2[1]<<16 |
            (SPC_i_0_0 & 0x0000ffff);
```

Note, that the expression (`*(int *)s`*) in a load or a store would lead to an implementation using the concatenation* `{SPC_s_0_2[0], SPC_s_0_2[1]}` *as in Example 3.*

Loads and stores from pointers to multiple locations are replaced by a set of assignments in which the locations are dynamically accessed according to the pointer's value.

The addresses (i.e. pointers' values) are encoded. The encoded value of a pointer `p` consists of two fields: the *tag* `p.tag` (left part of the code) corresponds to the location set referenced by the pointer and the *index* `p.index` (right part of the code) stores the number of bytes corresponding to the data referenced within the location set. After encoding, the size of the pointers can be reduced as shown in [9,10]. However, in order to support type casting and out-of-bound array accesses, we assume that pointers have a fixed size. In the rest of the paper, the size of the tag and the index are supposed to be equal to 16 bits.

The *index* part is stored within the first bits (least significant bits) of the code to support pointer arithmetic and type casting. An example for the implementation of an array of pointers is represented on Figure 3. It is important to note that, with this implementation, pointer arithmetic, even performed after type casting from pointer type to integer type, is straightforward to implement.

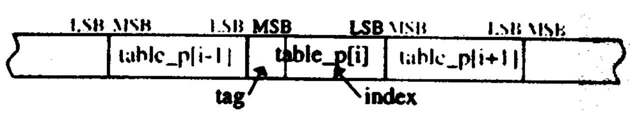

Figure 3: Encoding of pointers in an array

Loads and *stores* can then be removed using temporary variables and branching statements.

Example 5. *In the code segment below, the pointer* `p` *may point to the location sets* <its,0,2> *and* <b,0,4>.

```
int *p;
struct { int i; short ts[2]; } its;
int b[5];

if(...)
   p = &its.i;
else
   p = &b[2];
p = p+1;
out = *p;
```

The resulting code after removing the load and store is the following:

```
int SPC_p_0_0;
short SPC_its_0_2[4];
int SPC_b_0_4[5];

if(
   // p.tag = 0     // p index
   SPC_p_0_0 = 0 << 16   0;
else
   // p.tag = 1     // p index = 2 * 4
   SPC_p_0_0 = 1 << 16   32;

SPC_p_0_0 = SPC_p_0_0 + 4;     p = p + 1

if( SPC_p_0_0 >> 16 == 0 ) // (p.tag==0)
   // out = SPC_its_0_2[p.index/2]
   out = SPC_its_0_2[
          SPC_p_0_0&0xffff >> 1
         ];
else // (tag==1)
   // out = SPC_its_0_2[p.index/4]
   out = SPC_b_0_4[
           SPC_p_0_0&0xffff >> 2
```

The resolution of pointers can be further optimized. Loads and stores can be optimized when the pointers' location is a unique location set (i.e. case of a pointer variable) [9]. Encoding techniques [10] can also be to used to reduce the size of the pointers' value (*tag* part).

3.3 Dynamic memory allocation

In order to support dynamic memory allocation and deallocation, the hardware needs to access an allocator. In general, the allocator could be implemented in software (for mixed hardware/software implementations) or completely in hardware. Since this work is on the synthesis of hardware from C, we only consider a hardware implementation.

In software, `malloc` and `free` are implemented as standard library functions. Similarly, for hardware synthesis, we use a library of hardware components implementing `malloc` and `free`. The idea here is to have one component, called *allocator*, implement both the `malloc` and `free` functions. In order to efficiently manage memory, the memory space is partitioned into different *memory segments* in which data can be allocated.

Definition 1. *A* memory segment *is defined as an array of finite size in which data are allocated by a unique allocator. This array may later on be mapped to one or more memories during synthesis.*

In our tool, the mapping of heap objects to the different memory segments is done by the designer. Other tools could be used to assist this task at the system-level. For each `malloc` in the code, the designer selects in which memory segment the storage is allocated. Since the size of the dynamically allocated memory is a priori unknown at compile time, the designer also sets the size of each memory segment. The tool instantiates then the allocators corresponding to each memory segment and synthesizes the appropriate circuit to allocate, access and deallocate data.

For each memory segment, a different allocator is instantiated. Each `malloc` mapped to this memory segment is then replaced by a call to the specific allocator. The pointer that takes the result of the `malloc` function is defined as follows: its *tag* is set according to the corresponding memory segment and its *index* is set by the allocator. When multiple `malloc` calls are mapped to a single memory segment, the corresponding allocator is shared.

For a call `free(p)`, in the general case where the pointer `p` may point to multiple locations, the data to be deallocated may be in one memory segment or another depending on the value of the pointer `p`. We generate a branching statement in which the different allocators, corresponding to the different memory segments, may be called according to the pointer's *tag*. The pointer's *index* is then sent to the allocator to indicate which block should be deallocated. Loads, stores and addresses are resolved as shown in the previous section. Examples 6 illustrates how `malloc` and `free` calls are resolved while removing pointers.

Example 6. *Consider the following code segment.*

```
p = malloc(1);
out = *p;
free(p);
```

If `malloc` *is mapped to a memory segment called* `seg1` *of size 32 bytes, we generate the following code (*`SPC_p_0_0 & 0xffff` *implements* `p.index`*):*

```
char seg1[32]; // memory segment: seg1
SPC_p_0_0 = alloc_seg1(SPC_MALLOC,1);
SPC_out_0_0 = seg1[SPC_p_0_0 & 0xffff];
alloc_seg1(SPC_FREE,SPC_p_0_0 & 0xffff);
```

The allocator component corresponding to the function `alloc_seg1` *is called for both* `malloc` *and* `free`. *It implements both the allocation and deallocation functions.*

Further optimization can be performed [11]. The allocator architecture may be simplified by modifying the encoding of the pointers' value. Sequences of `malloc` and `free` may also be optimized.

4. IMPLEMENTATION AND RESULTS

4.1 Toolflow

We have implemented the different techniques presented here in our tool SpC using the SUIF environment [19]. The toolflow is shown on Figure 4. Our implementation takes a C function with complex data structures involving pointers and `malloc/free` and generates a Verilog module. The memory representation, consisting of distinct location sets, is used to map memory locations onto variables and arrays in Verilog. The resulting Verilog module can then be synthesized using the Behavioral Compiler of Synopsys.

In addition to the C input function, the designer defines a set of memory segments as well as the mapping of each `malloc` call to one of these memory segments. The `malloc/free` calls are then replaced by calls to the custom allocator function. Pointers are then resolved: loads and stores are replaced and pointers' values are encoded. During memory partitioning, locations represented by a unique location set are mapped to variables of fundamental type (e.g. `char`, `short`, `int`) and locations represented by a multiple location set are mapped to arrays derived from a fundamental type (e.g. array of `int`). Finally the resulting C code without pointers and structures gets translated into Verilog. Each type of allocator is defined as a hardware component in a library. During the translation into HDL, the different allocators corresponding to each memory segment are instantiated and the custom allocator functions are mapped to these allocator modules. The communication between each allocator and the main module is done using hand-shakes. The resulting HDL code can then be synthesized using traditional high-level synthesis tools.

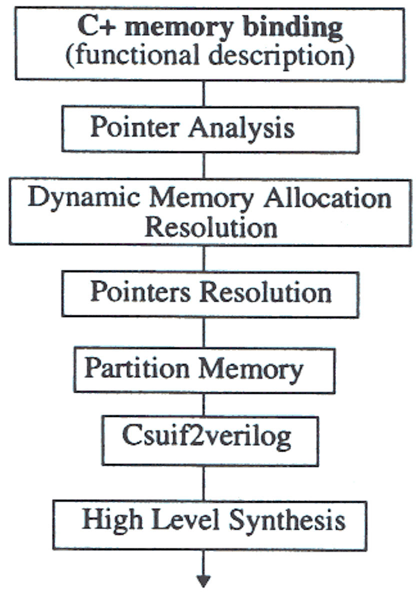

Figure 4: Resolution of dynamic memory allocation and pointers for hardware synthesis from C

4.2 Experimental results

We present two examples of implementations using SpC. The first example is a filter used in the JPEG library of Synopsys COSSAP [20] to perform, for example, RGB to YCrCb transformations. The filter implements the operation $Y[i] = clip(A \cdot X[i] + B, C)$ for $i = \{1, 2, ..., n\}$, where A is a 3×3 matrix, B and C are vectors, and X and Y are two $3 \times n$ dynamically-allocated matrix.

The second example is the implementation of an ATM segmentation engine. The segmentation engine receives frames to be sent from the host. These frames are segmented into 48 byte cells (payload of an ATM cell) to be transmitted on the network. The engine keeps track of each frame in a queue. For every new frame, a new virtual connection is open and a new queue element is allocated. As a results, we have two sets of `malloc` calls: one to allocate queue elements, the other to allocate connection status records.

For each example, we present two sets of results. The first set of results illustrates the case where `malloc` calls are mapped to two separate allocators (*no sharing*). In the second set of results, one allocator is shared (*sharing*). The allocators are taken from our library [11]. We use two types of allocators. *General-purpose* allocators can manage elements of arbitrary size. Blocks are allocated using a *first-fit* scheme. During deallocation adjacent free blocks are merged into larger blocks. *Specific-purpose* allocators, on the other hand, are much simpler. They can only allocate fixed-size element. Their allocation and deallocation schemes are then straightforward. The ATM segmentation engine may use either one general-purpose allocator or two specific-purpose allocators. Using two specific-purpose allocators is then preferable.

test	malloc/ free	C lines	number of allocators	HDL lines	size (1000x) comb.	size (1000x) non-c.	CPU time (in s)
jpeg	4 / 4	292	2 gen.-purp. (no sharing)	325	1,127	588	23.8
			1 gen.-purp. (sharing)	398	791	455	23.8
ATM seg.	4 / 2	403	2 spec.-purp. (no sharing)	618	430	334	34.6
			1 gen.-purp. (sharing)	611	557	325	33.8

Table 2: Results using one or two allocators (size in library units using the tsmc.35 target library; frequency 100MHz for ATM segmentation engine, 50MHz for JPEG; CPU time measured on Sun Ultra2 does not include high level synthesis)

5. CONCLUSION

We have presented an extension of the synthesizable C subset to pointers and complex data structures. In order to efficiently partition the storage among the different data sets during analysis and synthesis, memory is represented by *location sets*. During memory partitioning, locations represented by unique location sets are mapped to simple variables and locations represented by multiple location sets are mapped to arrays. Pointers are encoded and loads/stores are replaced by assignments in which data are directly accesses. Finally, dynamic memory allocation and deallocation are performed within each user-defined *memory segments* by an optimized hardware allocator instantiated from a library.

Our tool SpC, implemented within the SUIF compiler environment, takes a C function with pointers and complex data structures and generates a Verilog module which can be synthesized by commercial tools.

ACKNOWLEDGMENT

This work, done at Stanford University, was supported in part by Synopsys Inc. Koichi Sato was on leave from NEC Corporation.

REFERENCES

[1] Francky Catthoor, Sven Wuytack, Eddy De Greef, Florin Balasa, Lode Nachtergaele, Arnout Vandecappelle, *"Custom Memory Management Methodology,"* Kluwer Academic Publishers, Dordrecht, June 98.

[2] Abhijit Ghosh, Joachim Kunkel, Stan Liao, *"Hardware Synthesis from C/C++,"* in Proc. Design Automation and Test in Europe DATE'99, pp. 387-389, Munich, 1999.

[3] Andrew Kay, Toshio Nomura, Akihisa Yamada, Koichi Nishida, Ryoji Sakurai, and Takashi Kambe, *"Hardware Synthesis with Bach System,"* in Proc. IEEE International Symposium on Circuits and Systems ISCAS'99, Orlando, may 99.

[4] David Ku and Giovanni De Micheli, *"High-Level Synthesis of ASICs under Timing and Synchronization Constraints,"* Kluwer Academic Publishers, Boston, MA 1992.

[5] Luciano Lavagno, Ellen Sentovich, *"ECL: A Specification Environment for System-Level Design,"* in Proc. Design Automation Conf. DAC99, New Orleans, pp. 511-516, June 99.

[6] Giovanni De Micheli, *"Hardware Synthesis from C/C++,"* in Proc. Design Automation and Test in Europe DATE'99, pp. 382-383, Munich, 1999.

[7] Preeti Ranjan Panda, Nikil D. Dutt, Alexandru Nicolau, *"Memory Issues in Embedded Systems-On-Chip: Optimizations and Exploration,"* Kluwer Academic Publishers, October 1998.

[8] P. Schaumont, S. Vernalde, L. Rijnders, M. Engels, I. Bolsens, *"A Programming Environment for the Design of Complex High Speed ASICs,"* in Proc. Design Automation Conf. DAC'98, pp. 315-320, San Francisco, June 1998.

[9] Luc Séméria, Giovanni De Micheli, *"SpC: Synthesis of Pointers in C. Application of Pointer Analysis to the Behavioral Synthesis from C,"* in Proc. Int. Conf. on Computer-Aided Design ICCAD'98, pp. 340-346, San Jose, November 1998.

[10] Luc Séméria, Giovanni De Micheli, *"Encoding of Pointers for Hardware Synthesis,"* in Proc. Int. Workshop on IP-based Synthesis and System Design (IWLAS'98), pp 57-63, Grenoble, December 98.

[11] Luc Séméria, Koichi Sato, Giovanni De Micheli, *"Resolution of Dynamic Memory Allocation and Pointers for the Behavioral Synthesis from C,"* in Proc. Design Automation and Test in Europe DATE'00, Paris, March 2000.

[12] Charles Stoud, Ronald Munoz, David Pierce, *"Behavioral Model Synthesis with Cones"*, IEEE Design & Test of Computers, Vol 5 No3, pp.22-30, June 88.

[13] Kazutoshi Wakabayashi, *"C-based Synthesis with Behavioral Synthesizer, Cyber,"* in Proc. Design, Automation and Test in Europe DATE'99, pp. 390-391, Munich, 1999.

[14] Robert Wilson, Monica Lam, *"Efficient Context-Sensitive Pointer Analysis for C Programs,"* in Proc. of the ACM SIGPLAN'95 Conf. on Programming Languages Design and Implementation, pp.1-12, June 95.

[15] Sven Wuytack, Julio da Silva Jr., Francky Catthoor, Gjalt de Jong, Chantal Ykman, *"Memory Management for Embedded Network Applications,"* Trans. on Computer Aided Design, Volume 18, number 5, pp 533-544, May 99.

[16] CynApps Inc., http://www.cynapps.com

[17] Handle-C, http://oldwww.comlab.ox.ac.uk/oucl/groups/hwcweb/handel/

[18] SpecC, http://www.specc.gr.jp

[19] Suif Compiler, http://suif.stanford.edu

[20] Synopsys Inc. http://www.synopsys.com

[21] SystemC, http://www.SystemC.org