Synthesis of Hardware Models in C With Pointers and Complex Data Structures

Luc Séméria, Koichi Sato, and Giovanni De Micheli, *Fellow, IEEE*

***Abstract*—One of the greatest challenges in a C/C++-based design methodology is efficiently mapping C/C++ models into hardware. Many networking and multimedia applications implemented in hardware or mixed hardware/software systems now use complex data structures stored in multiple memories, so many C/C++ features that were originally designed for software applications are now making their way into hardware. Such features include dynamic memory allocation and pointers for managing data. We present a solution for efficiently mapping arbitrary C code with pointers and `malloc/free` into hardware. Our solution, which fits current memory management methodologies, instantiates an application-specific hardware memory allocator coupled with a memory architecture. Our work also supports the resolution of pointers without restriction on the data structures. We present an implementation based on the SUIF framework along with case studies such as the realization of a video filter and an ATM segmentation engine.**

***Index Terms*—Computer architecture, computer language, design automation, memory management, program compilers.**

I. Introduction

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 programming 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, and the possibility of leveraging a vast amount of legacy code and libraries, which facilitates the task of system modeling.

The use of C/C++ or a subset of C/C++ to describe both hardware and software accelerates the design process and facilitates software/hardware migration. Designers can describe their system using C/C++. The system can then be partitioned into software and hardware blocks, implemented using synthesis tools. The recent SystemC initiative[1] is an attempt to standardize a C/C++-based language for both hardware and software design.

The C language was originally designed to develop the UNIX operating system. It provides constructs to directly access memory (through pointers) and to manage memory and I/O using the standard C library (`malloc, free,` etc.). These constructs are widely used in software. Nevertheless, many of the networking and multimedia applications implemented in hardware or mixed hardware/software systems are also using complex data structures stored in one or multiple memory banks. As a result, many of the C/C++ features that were originally designed for software applications are now making their way into hardware.

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 [9], [15]. This task turns out to be particularly difficult because of dynamic memory allocation, function calls, recursion, `gotos`, type casting, and pointers.

In the recent past, different synthesis tools have been announced to ease the mapping of C code into hardware [6], [23]. All of these tools support a subset of the language (e.g., restrictions on pointers, function calls, etc.). In particular, they do not support dynamic memory allocation using the ANSI standard library functions `malloc` and `free`.

The overall objective of our research is to explore synthesis from full ANSI C. In our tool SpC [17], [19], pointer variables are resolved at compile time to synthesize C functional models in hardware efficiently. In this paper, we focus on the mapping of complex data structures into hardware. Specifically, we present how arrays of pointers as well as pointers inside of complex data structures can be efficiently mapped to hardware. In addition, a solution for the synthesis of dynamic memory allocation (`malloc/free`) is also presented. By definition, storage for dynamically allocated data structures cannot be assigned at compile time. The synthesis of C code involving dynamic memory allocation requires access to some allocation and deallocation primitives implemented either in software, as in an operating system, or in hardware.

Dynamic memory allocation is tightly coupled with pointers and the notion of a single continuous address space. Pointer dereferences (*load*, *stores*, etc.) as well as memory allocation are all referring to a main memory. However, in application-specific hardware, designers may want to optimize the memory architecture by using register banks, multiple memories etc. Therefore, memory allocations may be distributed onto multiple memories, and pointers may reference data stored in registers, memories, or even wires (e.g., output of a functional unit). To enable efficient mapping of C code with pointers and `mallocs` into hardware, the synthesis tool has to automatically generate the appropriate circuit to dynamically allocate, access (read/write), and deallocate data. Memory management as well as accurate pointers'

Manuscript received October 25, 2000. This work was supported in part by ARPA, MARCO Gigascale Research Center, and Synopsys Inc.

L. Séméria is with Clearwater Networks, Inc., Los Gatos, CA 95023 USA (e-mail: luc@clearwaternetworks.com).

G. De Micheli is with the Computer Systems Laboratory, Stanford University, Stanford, CA 95305 USA (e-mail: nanni@galileo.stanford.edu).

K. Sato is with the System LSI Design Engineering Division, NEC Corporation (e-mail: koichi@ax.jp.nec.com).

Publisher Item Identifier S 1063-8210(01)03357-1.

[1]See http://www.systemc.org/.

resolution are key features for C-based synthesis. They are enablers for the efficient design of applications involving complex data structures.

The contribution of this paper is to present a solution for efficiently mapping arbitrary C code with pointers and `malloc/free` into hardware. Our solution fits current memory management methodologies. It consists of instantiating a hardware allocator tailored to an application and a memory architecture. Our work also supports the resolution and optimization of pointers without restriction on the data structures.

In Section II, we give an overview of the memory-management methodology for embedded applications and present how it can be applied to the synthesis of hardware from C. The resolution of `mallocs` and pointers is based on an accurate analysis of the operations performed on the different memory locations. In Section III, we present our memory representation as well as some pointer-analysis techniques. Then, in Section IV, we show how pointers and dynamic memory allocation can effectively and efficiently be synthesized. In Section V, some optimizations are presented. We introduce our library of custom hardware memory allocators. Finally, in Section VI, we present an implementation and some results for different examples as well as the realization of a video filter and an asynchronous transfer mode (ATM) segmentation engine.

II. Methodology and Related Works

For decades, memory management has been one of the major development areas for both software and computer architecture. In software, at the user level, memory management is typically performed by the operating system. In hardware, memory bandwidth is often a bottleneck in applications such as networking, signal processing, graphics, and encryption. Memory architecture exploration and efficient memory management technology are key to the design of new high-performance systems. Memory generators commercially available today[2] enable fast integration of memories in a system. Scheduling of memory accesses has also been integrated into most commercial high-level synthesis (HLS) tools. Most of the refinement and compilation steps developed for software applications can also be used for hardware. Nevertheless, a software methodology usually assumes a fixed memory architecture, which may be general purpose or application specific, like in a digital signal processor or application-specific IP. In hardware, at the behavioral level, designers would typically explore different memory architectures in order to trade off area and power for a given timing constraint.

In the recent past, a few projects have been looking at means to use C/C++ as an input to current design flow [13]. 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 types. For reactivity, SystemC [11] from Synopsys, CoWare, and Frontier Design supports a mixed synchronous and asynchronous approach implemented as a C++ library. Other extensions include ECL [10] from Cadence based on C and Esterel, Handle-C[3] and Bach-C [7] originally based on Occam, SpecC[4] based on SpecChart, and CynLib.[5] 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 an approach include the early Bach-C compiler [7] from Sharp, OCAPI[6] [16] 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 CoCentric SystemC Compiler (formerly known as Scenic [6]) and by NEC with Cyber [23].

In practice, current tools do not support dynamic memory allocation and have restriction on pointers' usage [13]. SpC [17], [19] enables the behavioral synthesis of C code with pointer variables to variables and arrays. In Section IV, we present how pointers in general (e.g., array of pointers, pointers in structures, pointers to structures etc.) and dynamic memory allocation can also be efficiently synthesized.

A methodology for the design of custom memory systems has been described by Catthoor *et al.* [1]. It is defined for two sets of applications, networking and signal processing, and supports a limited subset of C/C++. The basic concepts presented in Catthoor's work can be generalized to support a larger subset of the C syntax for an extended set of applications. Two main steps can be distinguished in the methodology: we describe briefly here the transformations performed first at the system level, and then at the architectural level.

At the system level, the functionality of the algorithm is verified. Data formats are refined. For example, after quantization, the format of data can be refined from floating-point to fixed-point [8]. Data structures can also be refined for example to reduce the number of indirect memory references. Examples of such transformations for networking applications have been studied by Wuytack *et al.* [28].

At the architectural level, after partitioning, the system typically consists of multiple communicating processes to be mapped to hardware or software[7] [6], [21]. Memory segments are defined for internal storage and/or shared memory. These memory segments can then be mapped to one or multiple memories during synthesis. Some of the storage area (e.g., internal variables, etc.) can be statically allocated during synthesis or compilation. However, to support dynamic storage allocation (e.g., for recursive data structures), allocation and deallocation primitives implemented in software or hardware shall be defined.

In software, memory allocation and deallocation are implemented as primitives that are part of the operating system (OS). These primitives can be called in a C program using the functions defined in the standard library (e.g., `malloc, free,` etc.). Many schemes have been developed for OS to manage

[2]Silicon Access, DRAMatic: http://www.siliconaccess.com/

[3]See http://oldwww.comlab.ox.ac.uk/oucl/groups/hwcweb/handel/

[4]See http://www.specc.gr.jp/

[5]See http://www.cynapps.com/

[6]See http://www.imec.be/ocapi/

[7]See http://www.coware.com/

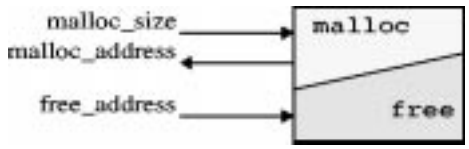

Fig. 1. Interface of the allocator block implementing `malloc` and `free` functions.

memory. An extensive survey by Wilson *et al.* [24] presents many of the techniques used for memory allocation and deallocation in software.

Memory management can also be implemented in hardware. For memory allocation and deallocation, instead of the system calls to the OS, requests are sent through signals to an *allocator* block (also known as a *virtual memory manager* [29]) implemented in hardware. Its interface is shown on Fig. 1. Internally, the allocator stores a list of the free blocks in memory as well as a list of the allocated blocks. To allocate memory, the size of the block to be allocated (*malloc_size*) is sent. The allocator then searches in its free list for a big enough block and returns the address of the beginning of this block (*malloc_address*). Two techniques are often used: *first fit* where the first acceptable free block is returned or *best fit* where the block of minimal size is returned. To free previously allocated memory, the address of the block to be deallocated (*free_address*) is sent to the allocator. The allocator then searches inside of the allocated list the block and adds it back to the free list. Adjacent free blocks can then be merged. An optimized architecture to speed up memory allocation in hardware is presented by Chang *et al.* [2]. The implementation itself of the allocator may also vary according to the application and the data structures. A number of these application-specific implementations are presented by Wuytack *et al.* [29].

Once an architecture is decided, hardware can be implemented using synthesis tools and compilers can be used for software. As far as memory management is concerned, scheduling of memory accesses, register/memory allocation, and address generation can be integrated into synthesis tools and compilers. In current commercial synthesis tools, each array is manually mapped to register files or memories of different types. For scheduling, the characteristics of the memories (number of ports, read/write latency, etc.) are defined as part of the components library. Researches have been looking at techniques to automatically perform memory assignment, address generation, and optimized scheduling according to the type of the memory. The latest development of these techniques have been presented by Catthoor *et al.* [1] and Panda *et al.* [14].

Our contribution fits in the methodology described above. In particular, we present techniques to automate the synthesis of C code with pointers and dynamic memory allocation into hardware. The outcome of our research is a tool that maps and optimizes hardware models in C into Verilog HDL synthesizable by commercially available synthesis tools.

III. Background

In software, a C program is targeted to a virtual architecture consisting of one memory in which all data are stored. The semantics of pointers is the address of an element in memory. 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 notions 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, a chip design contains multiple memory banks, register files, registers, and wires. To efficiently map C code onto hardware, the storage space must be partitioned. During synthesis, each partition is then mapped to a register, a wire, or a memory. Some of these partitions may also represent pointers. Pointers may be used to reference any variable no matter where its information is available. Pointers are then considered as references: references to memory elements, registers, wires, or ports. In particular, pointers can be used to allocate, read, write, and deallocate data. In this paper, we call the action of reading data using a pointer a *load*. Subsequently, a *store* is the action of writing data using a pointer. Allocation and deallocation are performed through the standard library functions `malloc` and `free`. Their implementation is, however, tailored for a given application and memory architecture.

The synthesis of hardware from C consists first of partitioning the memory. Each partition is then mapped to a scalar variable (akin to wire or register in the final implementation) or an array (akin to memory or register file). The synthesis of pointers consists of generating the appropriate circuit for allocating, accessing, and deallocating data. For this purpose, we change the addresses into numbers (i.e., encode pointers' values) and replace *loads* and *stores* by some assignments directly accessing the data the pointer may reference (i.e., resolve pointers). Functions `malloc` and `free` are subsequently changed, as memory allocation can be distributed onto multiple memories.

Example 1: Let us consider an application where a hardware block receives objects of different sizes and processes them. In the final implementation, after partitioning the memory, some of the intermediate data are stored in registers or memories. In this example, some of the objects received are copied into a register (`reg`). Some others are only used within this block and are stored in private memory (`local_RAM`). Finally, some larger objects may also be accessed by other blocks and are stored in a shared memory (`shared_RAM`).

```
int reg;
int *p;
struct {char type; int data; int data2;}
  object;
…
if(object.type == REG)
  p = &reg;
if(object.type == INTERNAL)
  // allocate memory in local_RAM
  p = malloc(4);
else
  // allocate memory in shared_RAM
  p = malloc(8);
…
```

```
// store in reg, local_RAM or shared_RAM
*p = object.data;
…
if(object.type != REG)
  // free storage in local_RAM or
  // in shared_RAM
  free(p);
```

In order to implement the store (`*p = object.data`), the tool has to schedule a write operation into the register `reg`, the memory `local_RAM`, or the memory `shared_RAM`. It also needs to instantiate the correct circuit (steering logic) to access these locations. For this purpose, we need to know at compile time the set of locations the pointer `p` may point to (points-to set).

To implement `free(p)`, assuming that the memories `local_RAM` and `shared_RAM` are each managed by a specific allocator, the tool also needs to schedule a deallocation operation on one allocator or the other. The points-to information for the pointer `p` is also necessary.

As we can see in Example 1, in order to efficiently map C code into hardware, we first need to partition the memory. In our implementation, memory is represented as a set of location sets, described in Section III-A.

Subsequently, to synthesize *loads*, *stores*, and `free` operations into hardware, we need to know at compile time the set of locations the pointers may reference (*points-to* information). 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 problem there is to find statements in the program that may read or write the same locations (aliasing problem). For this purpose, the *aliasing* information has to be determined between pointers. The points-to information and the aliasing information are equivalent and can be determined by recent analysis techniques called *pointer analysis* or *alias analysis*, described in Section III-B.

A. Memory Representation

The simplest memory representation consists of a single address space in which all data are stored. This trivial representation, however, prevents optimizing the locality and parallelizing the code. On the other hand, the most accurate representation, which would distinguish each element of arrays or of recursive data structures, is not practical. As a result, most analysis techniques combine elements within a single data structure. Some techniques combine elements based on their allocation contexts [25], [26] or on limiting the length of access paths to some fixed constant (k-*limiting*). Shape analysis [3], [5] gives the most accurate representation, as it may distinguish trees from direct acyclic graphs, linear lists from cyclic lists, and so on. However, its implementation to support large C programs remains challenging.

In order to find both an accurate and a practical representation for hardware synthesis, we use the notion of *location sets* introduced by Wilson and Lam [25], [26]. Location 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.

TABLE I
LOCATION SET EXAMPLES (f = OFFSET OF FIELD F), (s = STRIDE OR ARRAY ELEMENT SIZE)

Type	Expression	Location Set
`int a`	`a`	$\langle a, 0, 0\rangle$
`struct {int F;} r`	`r.F`	$\langle r, f, 0\rangle$
`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]`	$\langle r, f \bmod s, s\rangle$

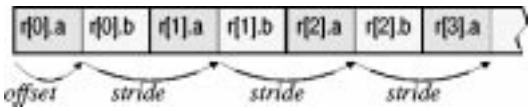

Fig. 2. Representation of `struct{int a; int b;} r[];` the offset and stride correspond to the locations `r[i].b` where `i` is integer.

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 $\{f + i.s | i \in \mathbb{Z}\}$ 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 I shows the location sets for various expressions.

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. Fig. 2 gives an example of representation for an array of structures. The representation does not 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.

Nested arrays and structures, type casting, and pointer arithmetic are making things more complicated, leading to some additional inaccuracies. Example 2 shows how references to array nested in structures are represented approximately. 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 fields of the structure by using out-of-bound array indexes.

Example 2: Consider the array `r.F[]` nested in a structure `r`:

```
struct {
  char a;
  char b;
  int F[8];} r;
```

References to one of the array elements (e.g., `r.F[2]`) are represented approximately by the locations set $\langle r, 0,$ `sizeof(int)`$\rangle$, which regroups all of the elements of the array as well as `r.a`.

Dynamically allocated memory locations (also known as heap-allocated objects) are represented by a specific location set. As far as accuracy, it would not be practical to distinguish every element of a recursive data structure. Therefore, the goal of this representation is to distinguish complete data structures. The different elements of a recursive data structure would typically be combined into one location set. For example, we want to distinguish one list from another but we do not want to distinguish the different elements of a list. Heuristics are used to distinguish dynamically allocated data. Storage allocated in the same context is assumed to be part of the same equivalence class. Within one function, storage allocated by a given `malloc` in the code is represented by one location set. When `malloc` is called inside of a function, a different location set is created for each call chain (context). These heuristics have been proven to work well as long as the program uses the standard memory allocation routines [25].

Example 3: In the code segment shown in Example 1, the memory can be represented by the following set of location sets: ⟨`p`, 0, 0⟩, ⟨`reg`, 0, 0⟩, ⟨`object`, 0, 0⟩ for `object.type`, ⟨`object`, 4, 0⟩ for `object.data`, ⟨`object`, 8, 0⟩ for `object.data2`, ⟨*malloc1*, 0, 0⟩ for the storage allocated by the first `malloc` call (`malloc`(4)), and ⟨*malloc2*, 0, 0⟩ for the storage allocated by the second malloc call (`malloc`(8)).

B. Pointer Analysis

Pointer analysis is a compiler technique to identify at compile-time the potential values of the pointers in the program. This information is used to determine the set of locations to which the pointer may point. For synthesis, in the case of *loads*, *stores*, and `free`, we want to synthesize the logic to access, modify, or deallocate the location referenced by the pointer. For this purpose, the points-to information must be both *safe* and *accurate*: safe because we have to consider all locations the pointer *may* reference and *accurate* because the smaller the points-to set is, the fewer logic circuits we have to generate.

Two main types of analyses can be distinguished. First, *flow-* and *context-insensitive* analyses [22] do not distinguish the order in which the statements are executed (*flow-insensitivity*) and the different calls of a function (*context-insensitivity*). They are the least accurate, but the relative simplicity of their implementation makes them more suitable for very large programs. *Flow- and context-sensitive* analyses, such as the one developed by Wilson and Lam [25], [26], on the other hand, provide more accuracy with an increased complexity.

Even though the computational 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 recursion. Besides, these analyses lead to more accurate results, which makes them more suitable for hardware synthesis.

Example 4: In the code segment presented in Example 1, annotations are inserted by the pointer analysis to specify what pointers may point to at loads, stores, and `free` calls.

```
if(object.type == REG)
  p = &reg; // ⟨p, 0, 0⟩ –> {⟨reg, 0, 0⟩}
if(object.type == INTERNAL)
  p = malloc(4); // ⟨p, 0, 0⟩ –> {malloc1}
else
  p = malloc(8); // ⟨p, 0, 0⟩ –> {malloc2}
…
*p = object.data; // ⟨p –> {reg, malloc1,
  malloc2}
…
if(object.type != REG)
  free(p); // p –> {reg, malloc1,
  malloc2}
```

In the previous code segment, the notation ⟨`p`, 0, 0⟩ `–>` {⟨`reg`, 0, 0⟩, {`malloc1`, `malloc2`} stands for "p may point to variable `reg` or some storage allocated by *malloc1* (fist malloc call) or *malloc2* (second malloc call)."

C. Definition of the Subset

In this section, we only talk about the restrictions on the synthesizable subset. Limitations on the generated architecture may also exist akin to the limitations of the behavioral synthesis tool used as a back end to our tool. In particular, the techniques presented here do not depend on the type of memories (SRAM, DRAM, etc.).

The pointer analyses and memory representation presented in the previous sections 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. The code is assumed to be correct. Tools such as Purify[8] or LCLint[9] [4] can be used to find memory leaks and check memory reads, writes, and deallocations. In addition, 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. Their resolution would require the synthesis of some kind of interface between the circuits realizing the processes. Such interface is usually defined during system partitioning and, hence, before synthesis. As a result, memory allocated in one process is assumed to be accessed and deallocated only within this same process.

The second limitation stems from the fact that most commercial synthesis tools also have restrictions on functions. Recursion is usually not supported. Procedures that are mapped to components typically have restrictions both on their functionality and their parameters. For example, the same function called within different contexts may usually not be shared. Besides, most synthesis tools do not synthesize parameter passed by reference, because this is not supported by most HDL syntax.

[8]See http://www.rational.com/

[9]See http://lclint.cs.virginia.edu/

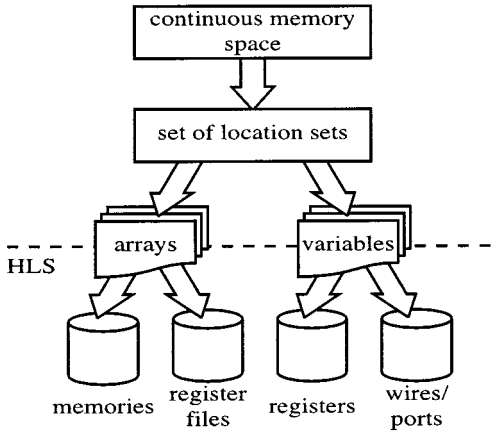

Fig. 3. Memory refinement from a continuous memory space to a set of memories, registers, and wires

The synthesis of functions in C, and therefore the resolution of pointers and `malloc/free` inside of functions, is beyond the scope of this paper.

IV. Mapping to Hardware

In this section, we present how C code can be efficiently mapped onto hardware. First, memory is partitioned into a set of location sets, which can be mapped onto wires, registers, or memories. Some of these location sets represent pointers. Pointers are resolved by encoding the value of the pointers and creating branching statements for loads and stores. Finally, dynamic memory allocation and deallocation are performed by custom hardware memory allocators.

A. Memory Refinement

After analysis, the storage in the program can be represented as a set of distinct locations sets. This set of location sets represents a partitioning of the memory. Each location set is ultimately mapped to a wire, a register, or a section of memory in the final design, as shown in Fig. 3. The allocation of a given scalar variable to a register or a wire as well as the mapping to memories or register files are typically the result of HLS. In this section, we present how distinct location sets can be mapped to a set of arrays and variables. We do not consider pointers and heap objects. The synthesis of pointers and `malloc/free` is presented in Sections IV-B and IV-C. In the rest of this 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-bit architecture. Derived types such as pointers, arrays, and structures are constructed from these fundamental data types.

We can distinguish two types of location sets for statically allocated data: location sets whose strides are null (i.e., singletons, sets of one location) and location sets with nonzero strides (i.e., sets of multiple locations). A singleton location set may therefore be treated as a simple variable, whereas a location set with nonzero stride may be mapped to an array. In our implementation [20], for each location set $\langle loc, f, s\rangle$, we define SPC_loc_f_s as follows.

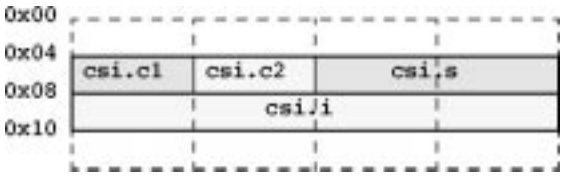

Fig. 4. Memory layout of `struct {char c1; char c2; short s; int i;} csi`.

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

For a location set with nonzero stride (i.e., s not null), SPC_loc_f_s is defined as an array (e.g., array of integers). Such an array may then typically either be mapped to a memory or a register file manually or according to current methodology [1], [14] during high-level synthesis. 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 one to independently access the different fields of the structures, leading to more flexibility and potentially better performances.

Example 5: 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 in Fig. 4.

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 6, a structure is represented as two integers. In Example 7, an integer inside of a structure is represented by the concatenation of two short integers.

Example 6: 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 7: 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_2[0] = SPC_a_0_0 > > 16;
SPC_its_0_2[1] = SPC_a_0_0 & 0xffff;

// b = its.i;
SPC_b_0_0 = SPC_its_0_2[0] < < 16 |
  SPC_its_0_2[1];
```

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

```
{SPC_my_str_0_2[0], SPC_my_str_0_2[1]} =
  SPC_a_0_0;
SPC_b_0_0 = {SPC_my_str_0_2[0],
             SPC_my_str_0_2[1]};
```

B. Pointers

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

In hardware, as discussed in Section III, data my 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 but may also reference registers, wires, or ports. Our synthesis tool generates the appropriate circuit to dynamically access these locations according to the pointers' value. The implementation presented here is a generalization of previous work [17], [18] to deal with complex data structures such as arrays of pointers, pointers within structures, type casting, and dynamic memory allocation.

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 within 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 6 and 7, the locations accessed may correspond to the concatenation of multiple elements of location sets. Moreover, because of pointer type casting, the location on which the load or store is performed may correspond to only part of an element of a location set, as shown in Example 8.

Example 8: 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;
*(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_i_0_0 & 0xffff)|
            SPC_s_0_2[1]≪16;
```

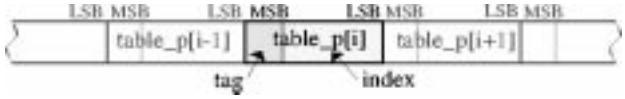

Fig. 5. Encoding of pointers in an array.

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 7.

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 (tag part) can be reduced as shown in [17], [18]. However, in order to support type casting and out-of-bound array accesses, we assume that pointers have a fixed size of 32 bits. In the rest of this 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 of implementation for an array of pointers is represented on Fig. 5. 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.

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

Example 9: 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 = 0
  SPC_p_0_0 = 0 < <16 | 0;
else
  // p.tag = 1 // p.index = 2 * 4
  SPC_p_0_0 = 1 < < 16 | 8;

SPC_p_0_0 = SPC_p_0_0 + 4; // p = p + 1;

if( SPC_p_0_0 > > 16 == 0 ) //
  (p.tag==0)
  // out = {SPC_itc_0_2[p.index/2],
  //        SPC_itc_0_2[p.index/2+1]};
  out = SPC_its_0_2[
          SPC_p_0_0&0xffff > >1
          ] ≪ 16 |
         SPC_its_0_2[ SPC_p_0_0&0xffff > >
  1 + 1];
else // (tag==1)
  // out = SPC_itc_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 set has a stride null (i.e., case of a scalar variable) [17]. Encoding techniques [18] can also be to used reduce the size of the pointers' value (*tag* part).

C. Resolution of `malloc` and `free`

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. For example, in a mixed hardware/software implementation, the hardware can send an interrupt to the processor to perform memory allocation and deallocation. In the case of an allocation, the software, typically the operating system running on the processor, would then send the address of the allocated block in memory to the hardware. Since this paper is on the hardware synthesis of C code, only a hardware implementation is presented. Nevertheless, the techniques presented here could also be targeted to a software 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,* implementing both the `malloc` and `free` functions, as introduced in Section II. In order to efficiently manage memory, we want to support multiple customized memory allocators that may allocate storage in multiple memories in parallel. As a result, we partition the memory space in which data are dynamically allocated (heap space) into a set of *memory segments* (pools of blocks).

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 high-level synthesis.

In our tool, the partitioning of the memory into the different memory segments is done by the designer. Other tools could be used to assist this task at the system level. For example, tools

such as the one defined in the Matisse research project[10] [28], [29] could be used in order to refine data structures and define different arrangements and architectures of hardware memory allocators.

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 cannot be found by static analysis, the designer also sets the size of each memory segment manually. The tool instantiates then the hardware memory 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)`, the data to be deallocated may be in one memory segment or another depending on the value of the pointer `p`. We generate branching statements in which the different allocators corresponding to the different memory segments may dynamically 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 10 and 11 illustrate how `malloc` and `free` calls are resolved while removing pointers.

Example 10: 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 (assuming that the size of `char` is one byte):

```
char seg1[32]; // memory segment: seg1
p_0_0 = alloc_seg1(SPC_MALLOC,1);
out_0_0 =
  seg1[p_0_0|0xffff]; // seg1[p.index]
alloc_seg1(SPC_FREE,p_0_0);
```

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

Example 11: Let us now consider a more complex example where pointer `p` may point to different memory segments:

```
if(i==0)
  p = malloc(1); // malloc1
else
  p = malloc(4); // malloc2
```

[10]See http://www.imec.be/matisse/

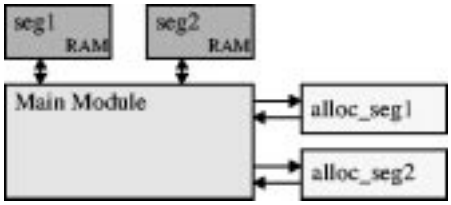

Fig. 6. Architecture for multiple memories and allocators.

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

We assume that *malloc1* is mapped to the memory segment *seg1* and *malloc2* is mapped to the memory segment *seg2*. Both memory segments are of size 32 bytes (set by the user). The resulting code, after removing `malloc/free`, is the following:

```
char seg1[32];
char seg2[32];
if(i==0) {
  p_0_0 = alloc_seg1(SPC_MALLOC,1);
} else {
  p_0_0 = alloc_seg2(SPC_MALLOC,4);
}
...
if(p_0_0 > > 16==0) // p.tag==0
  out_0_0 =
  seg1[p_0_0&0xffff]; // seg1[p.index]
else
  out_0_0 =
  seg2[p_0_0&0xffff]; // seg2[p.index]
...
if(p_0_0 > > 16==0) // p.tag==0
  alloc_seg1(SPC_FREE,p_0_0);
else
  alloc_seg2(SPC_FREE,p_0_0);
```

If each memory segment is mapped to a different RAM during synthesis, we end up with the architecture shown in Fig. 6.

V. Library of Allocators and Optimizations

In the previous sections, we have seen how `malloc` and `free` can be implemented using hardware memory allocators. Each allocator can perform both memory allocation and deallocation. We provide a library of such allocators. The designer has then the freedom to pick the allocator architecture most suitable to the application. Our library of allocator components contains three basic types of allocators. In Section V-A, we define as *general-purpose* an allocator that can allocate blocks of any size. In addition, we introduce here an *optimized general-purpose allocator*, for which the deallocation scheme is optimized for latency. When the size of the block to be allocated is a fixed constant, the architecture of the allocator can be greatly simplified. The *specific-purpose allocator* presented in Section V-B can be used in such case.

The designer could also add new allocators in the library. The basic allocators presented here may be modified (e.g., to change

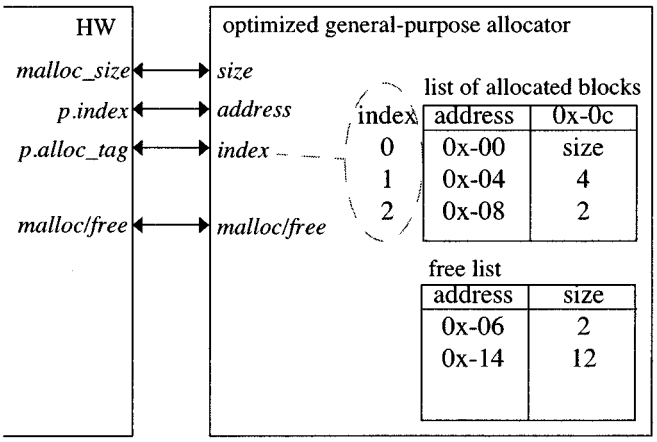

Fig. 7. Architecture of an optimized general-purpose allocator.

the allocation or deallocation schemes, allocate a larger number of blocks, or handle new sizes of elements) and added to the library. Other types of allocators, such as those described by Chang *et al.* [2] and Wuytack *et al.* [29], could also be added to our framework as new components in the library.

In some cases, the code can also be optimized. Calls to `malloc` and `free` can be removed and memory allocation can be done statically. In Section V-C, we present a compiler technique to automatically remove some of the dynamic memory allocations for sequences of `malloc` and `free`.

A. Optimized General-Purpose Allocator

General-purpose allocators are defined as allocators that may allocate blocks of various sizes. These allocators consist of the circuit that performs allocation/deallocation and two lists that keep track of the free blocks and the allocated blocks inside of the memory segment. To allocate memory, the size of the block to be allocated (*malloc_size*) is sent to the allocator. The allocator then searches in its free list a big enough block and returns the address corresponding to the beginning of this block in the memory segment. In our implementation, the first acceptable free block is returned (first fit). The block that has just been allocated is then added to the list of allocated blocks. To free previously allocated memory, the address of the block to be deallocated is sent to the allocator. The allocator then searches this block inside of its list of allocated blocks and adds it back to the free list. Adjacent free blocks are then merged.

In order to simplify the process of looking up for a given block during deallocation, we propose to encode the characteristics of the allocated block inside of the pointer's *tag*. In our implementation shown in Fig. 7, the allocator stores the list of allocated blocks in an array. The index corresponding to the allocated block in this array is then encoded in the pointer's value. During deallocation, this index is sent to the allocator. The allocator can then directly find the allocated block according to this index, without having to search the entire array. The resulting optimized allocator is called *optimized general-purpose*.

The encoded value of a pointer consists then of three fields: the *allocation tag*, the *tag*, and the *index*. For a pointer `p`, the *tag* `p.tag` and the *index* `p.index` are defined as in Section IV-B. The *allocation tag* `p.alloc_tag` corresponds to the index of

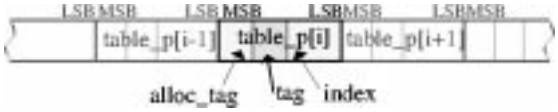

Fig. 8. Encoding of pointers in an array for optimized general-purpose allocator.

the block inside of the list of allocated blocks. In our implementation, the allocation tag corresponds to the 8 most significant bits in the pointer's value, the tag corresponds to the following 8 bits, and the index corresponds to the 16 least significant bits (as defined in Section IV-B). Fig. 8 shows how the different fields are laid out for an array of pointers.

B. Specific-Purpose Allocator

The `malloc` function takes one argument: the size of the block to be allocated. When this size is a unique constant K for all `malloc` calls mapped one memory segment, this memory segment can then be represented as an array of elements of size K. Allocating memory in this segment can simply be performed by returning the first available element in the array. For deallocation, the array element to deallocate can easily be derived from the block address. The architecture of the corresponding allocator can then be simplified. For example, a simple bit vector can be used to keep track of the allocated and free blocks in the memory segment. Such an allocator, which can only deal with blocks of one size, is called *specific-purpose*. Using a specific-purpose allocator solves also the problem of memory fragmentation common to general-purpose allocators.

Constant propagation can be performed before selecting the allocator in order to have as many `malloc`s as possible with constant size.

C. Optimizing Sequences of `malloc` and `free` Calls

Some of the dynamic memory allocations are sometimes not necessary and can be automatically removed at compile time. This is especially true for legacy code, in which `malloc/free` are used to manually control storage. The idea here is to analyze to code and isolate the finite sequences of `malloc` calls that can be replaced by references to statically allocated data.

Example 12: Consider the following code segment:

```
p[1] = malloc(4); // malloc1
p[2] = malloc(8); // malloc2
...
free(p[1]); // free1
free(p[2]); // free2
```

In this example, a finite number of objects (two) are allocated by *malloc1* and *malloc2*. Later on, these blocks are freed by *free1* and *free2*. The dynamic memory allocation in this case can be optimized by creating the two temporary array elements `tmp_malloc1`[4] and `tmp_malloc2`[8]. The size of these elements corresponds to the size of the object allocated at each `malloc`. The `malloc` calls are then replaced by references to these temporary variables and the `free` calls are removed. We

end up with the following code segment in which memory is statically allocated:

```
char tmp_malloc1[4];
char tmp_malloc2[8];
p[1] = tmp_malloc1; // malloc(4)
p[2] = tmp_malloc2; // malloc(8)
...
// free(p[1]);
// free(p[2]);
```

The optimization can be performed under two conditions. First, the size of the block to allocate has to be constant. If the size of the block to allocate is not known at compile time, a *general-purpose* or *optimized general-purpose* allocator would have to be used. Second, if a block is allocated within an unbounded loop, it has to be deallocated within the same unbounded loop. Using the results of the pointer analysis, we have implemented a dataflow analysis [12] that finds at compile time the `malloc` and `free` calls that can be optimized (i.e., removed).

We outline briefly how the analysis is conducted. For each dynamically allocated location set (i.e., each `malloc` call in the example), a counter is defined. The analysis steps through the flowgraph of the procedure. The counter is incremented each time an element of the corresponding location set is allocated. Subsequently, each time an element of the location set is deallocated (result from the pointer analysis), the associated counter is decremented. Location sets allocated and not deallocated within the same loop can be found. The `malloc` and `free` corresponding to these locations cannot be optimized. Otherwise, they can be optimized.

During the optimization, a temporary variable is created for each `malloc` that can be removed. The size of each temporary variable corresponds to the size in the `malloc` call. These temporary variables are then statically allocated during synthesis. The corresponding `free` calls are removed.

VI. Implementation and Results

A. Toolflow

In the previous sections, we have shown how pointers, `malloc/free`, and complex data structures can be resolved at compile time. A methodology to efficiently map C code onto hardware was also presented. At the system level, data structures are refined and a memory architecture is defined. At the architectural level, the system consists of a set of communicating processes. Each of these processes can then be mapped to software or hardware.

Several tools can be used in this methodology. For memory management at the system level, tools are used to help in refining data structures [28] and defining a memory architecture. Once an architecture is defined, high-level synthesis tools can be used to map functionality onto hardware. These high-level synthesis tools may also perform memory assignment, address generation, and scheduling of memory accesses [1], [14]. Our toolflow is shown on Fig. 9. Our tool SpC takes a C function with complex data structures and generates a Verilog module. As a front end to high-level synthesis, it is the first step for mapping C code involving pointers and dynamic memory allocation onto hardware. SpC can be seen as a back end to the system-level tools presented above. Such tools can help define a memory architecture and an arrangement of hardware memory allocators (*memory binding*).

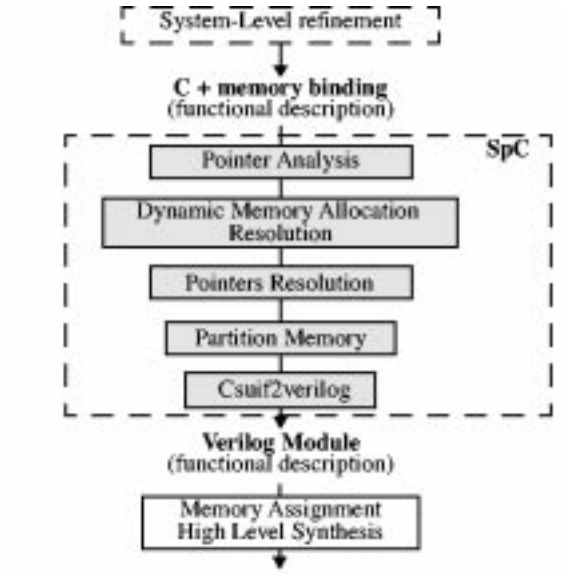

Fig. 9. Resolution of dynamic memory allocation and pointers for hardware synthesis from C.

The different techniques presented in Sections IV and V have been implemented using the SUIF compiler environment[11] [27]. 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, a set of memory segments as well as the mapping of each `malloc` call to one of these memory segments (*memory binding*) must be defined. We first try to remove some of `malloc/free` calls using the optimization in Section V-C. The remaining `malloc/free`s are then replaced by calls to a custom allocator function (*specific-purpose, general-purpose, or optimized general-purpose*). During memory partitioning, locations represented by location sets with a stride null are mapped to variables of fundamental type (e.g., `char, short, int`), and locations represented by location sets with nonzero stride are mapped to arrays derived from a fundamental type (e.g., array of `int`). Pointers are removed and the code gets translated into Verilog. Each type of allocator is defined as a hardware component in our 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 handshakes. The resulting HDL code can then be synthesized using traditional high-level synthesis tools.

[11]See http://suif.stanford.edu/

TABLE II
IMPLEMENTATION OF THE DIFFERENT ALLOCATORS [AREA IN LIBRARY UNITS USING THE TSMC.35 TARGET LIBRARY; *comb.* AND *noncomb.* REPRESENT, RESPECTIVELY, THE AREA OF COMBINATIONAL LOGIC AND NONCOMBINATIONAL LOGIC (i.e., REGISTERS, etc.) AT 100 MHz]

allocator	lines		size	
	C	HDL	comb.	non-comb.
general purpose	297	353	204,191	80,193
general purpose (opt)	289	349	212,065	81,652
specific purpose	85	135	33,579	19,830

We have recently ported our research to the Synopsys Cocentric SystemC Compiler to synthesize C models into hardware directly, without having to translate C into HDL.

B. Experimental Results and Discussion

For the set of examples presented here, we have synthesized three types of allocators in our library. In the results presented in Table II, allocators are designed to allocate up to 16 blocks of memory. They are synthesized directly from C using SpC and Synopsys Behavioral Compiler. The general-purpose allocators use *first-fit* to allocate blocks and merge adjacent free blocks during deallocation. The first row presents the results for the *general-purpose allocator* without any optimization. The second row shows the size of the *optimized general-purpose allocator* for which the deallocation scheme has been optimized using the modified *tag*, as presented in Section V-A. Even though the complexity of the controller is reduced (from 52 states to 46), the size of the optimized allocator is roughly the same because of an increase in the steering logic. The latency of the deallocation task is, however, reduced as shown later in Table III. Finally, the third row presents the results for the *specific-purpose allocator* introduced in Section V-B. As expected, its size is much smaller than the *general-purpose allocators*.

Table III shows the results for four different examples. The first two examples *test1* and *test2* consists of three `malloc` calls and two `free` calls. All `malloc` calls allocate objects of the same constant size. Hence a *specific-purpose allocator* can be used. For the first example, all calls to `malloc` and `free` can be removed during optimizations. For the second example, one of the `malloc`s is called inside of an unbounded loop and cannot be removed. The third example is a filter used in the JPEG library of Synopsys COSSAP and is used, for example, for RGB to YCrCb transformations. The filter implements the operation $\boldsymbol{Y}[i] = clip(\boldsymbol{A} \cdot \boldsymbol{X}[i] + B, C)$ for $== \{1, 2, \ldots, n\}$, where $\boldsymbol{A}$ is a 3×3 matrix, B and C are vectors, and $\boldsymbol{Y}$ and $\boldsymbol{X}$ are two $3 \times n$ dynamically allocated matrix. Finally, the last 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 opened and a new queue element is allocated. As a result, we have two sets of malloc calls: one to allocate queue elements and the other to allocate connection status records.

TABLE III
RESULTS FOR THE DIFFERENT EXAMPLES AND OPTIMIZATIONS (SIZE IN LIBRARY UNITS USING THE tsmc.35 TARGET LIBRARY; FREQUENCY 100 MHz FOR TEST1, TEST2 AND ATM, 50 MHz FOR JPEG; CPU TIME FOR SYNTHESIS MEASURED ON SUN ULTRA2 DOES NOT INCLUDE HIGH-LEVEL SYNTHESIS)

test	malloc /free	C lines	optimization	HDL lines	total latency (in ns)	size (1000x)		CPU time (in s)
						comb.	non-c.	
test1	3 / 2	72	gen. alloc. (no sharing)	344	713	568	269	14.8
			gen. alloc.	315	735	391	180	13.8
			gen. alloc. (optimized)	323	617	405	199	14.4
			sequence	167	32	135	87	14.3
test2	3 / 2	66	gen. alloc. (no sharing)	339	1,425	551	271	13.8
			gen. alloc.	310	1,732	338	177	13.4
			gen. alloc. (optimized)	318	1,221	372	177	13.2
			spec alloc.	294	781	190	109	12.9
jpeg	4 / 4	190	gen. alloc. (no sharing)	659	438	1,287	747	21.7
			gen. alloc.	630	465	1,023	632	20.6
			gen alloc (optimized)	640	403	1,025	637	20.6
ATM	4 / 2	403	spec alloc. (no sharing)	618	551	508	419	35.3
			gen. alloc.	611	904	1,359	693	35.3
			gen alloc (optimized)	574	696	1,055	547	35.3

For each example, the first set of results illustrates the case where `malloc` calls are mapped to two *general-purpose* allocators (*no sharing*). For the ATM segmentation engine, two *specific-purpose* allocators are used instead of the *general-purpose* allocators. In the other results, one allocator is shared. As expected, the latency (measured by simulation at the RTL level) increases without sharing with a decrease in area. In Table III, we can also verify that the total latency of the design decreases when the *optimized general-purpose allocator* (*gen. alloc. optimized*) is used. The use of a *specific-purpose allocator* (*spec. alloc.*) when possible provides significant reduction both in latency and area. Finally, further optimizations can be performed when sequences of `malloc` and `free` calls can be removed (*sequence*).

VII. CONCLUSION

We have presented how C code with pointers and `malloc/free` can be efficiently mapped to hardware. With our methodology, memory is partitioned into a set of location sets and pointer analysis is used to define which locations are accessed and deallocated in the program. Pointers are synthesized by encoding their values and generating circuits to dynamically access the different locations they may reference.

Dynamic memory allocation and deallocation are implemented using one or multiple hardware allocators.

Our toolflow fits into current memory management methodology. High-level synthesis is used to map data to multiple memories, registers, and wires. Different schemes for allocating and deallocating memory are also supported by adding hardware memory allocators in our library of allocators. As part of this library, we have presented an optimized architecture for a general-purpose allocator. This optimization consists in encoding the characteristics of the allocated block referenced as part of the pointer's value to speed up deallocation. When the size of the block to allocate is a fixed constant, a specific-purpose allocator may also be used to optimize both area and latency.

The synthesis of pointers and `malloc/free` raises the level of abstraction at the input of high-level synthesis. It facilitates the description and implementation of custom memory architectures. Models can be described at the behavioral level using the notions of a single address space and indirect memory references found in many programming languages. The techniques presented here can be generalized to support more of the C/C++ syntax as well as other programming languages, enabling the mapping of functions and complex data structures including object-oriented features into hardware.

REFERENCES

[1] F. Catthoor, S. Wuytack, E. De Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle, *Custom Memory Management Methodology*. Dordrecht, Germany: Kluwer Academic, June 1998.

[2] J. M. Chang and E. R. Gehringer, "A high-performance memory allocator for object-oriented systems," *IEEE Trans. Comput.*, vol. 45, Mar. 1996.

[3] A. Deutsh, "Interprocedural may-alias analysis for pointers: Beyond k-limiting," in *Proc. ACM SIGPLAN'94 Conf. Programming Language Design and Implementation*, June 1994, pp. 230–241.

[4] D. Evans, "Static detection of dynamic memory errors," in *Proc. SIGPLAN Conf. Programming Language Design and Implementation (PLDI '96)*, Philadelphia, PA, May 1996.

[5] R. Ghiya and L. Hendren, "Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C," in *Proc. 23th Annu. ACM Symp. Principle of Programming Languages*.

[6] A. Ghosh, J. Kunkel, and S. Liao, "Hardware synthesis from C/C++," in *Proc. Design, Automation and Test in Europe DATE'99*, Munich, Germany, 1999, pp. 387–389.

[7] A. Kay, T. Nomura, A. Yamada, K. Nishida, R. Sakurai, and T. Kambe, "Hardware synthesis with Bach system," in *Proc. IEEE Int. Symp. Circuits and Systems ISCAS'99*, Orlando, FL, May 1999.

[8] H. Keding, M. Willems, M. Coors, and H. Meyr, "FRIDGE: A fixed-point design and simulation environment," in *Proc. Design Automation and Test in Europe DATE'98*, 1998, pp. 429–435.

[9] B. Kernighan and D. Ritchie, *The C Programming Language*. Englewood Cliffs, NJ: Prentice-Hall, 1988.

[10] L. Lavagno and E. Sentovich, "ECL: A specification environment for system-level design," in *Proc. Design Automation Conf. DAC'99*, New Orleans, LA, June 1999, pp. 511–516.

[11] S. Liao, S. Tjang, and R. Gupta, "An efficient implementation of reactivity for modeling hardware in the scenic design environment," in *Proc. Design Automation Conf. DAC'97*, June 1997, pp. 70–75.

[12] S. Muchnick, *Advanced Compiler Design and Implementation*. San Francisco, CA: Morgan Kaufmann, 1997.

[13] G. De Micheli, "Hardware synthesis from C/C++," in *Proc. Design, Automation and Test in Europe DATE'99*, Munich, Germany, 1999, pp. 382–383.

[14] P. R. Panda, N. D. Dutt, and A. Nicolau, *Memory Issues in Embedded Systems-on-Chip: Optimizations and Exploration*. Norwell, MA: Kluwer Academic, 1998.

[15] P. J. Plauger, *The Standard C Library*. Englewood Cliffs, NJ: Prentice-Hall, 1991.

[16] P. Schaumont, S. Vernalde, L. Rijnders, M. Engels, and I. Bolsens, "A programming environment for the design of complex high speed ASICs," in *Proc. Design Automation Conf. DAC'98*, San Francisco, CA, June 1998, pp. 315–320.

[17] L. Séméria and G. De Micheli, "SpC: Synthesis of pointers in C. Application of pointer analysis to the behavioral synthesis from C," in *Proc. Int. Conf. Computer-Aided Design ICCAD'98*, San Jose, CA, Nov. 1998, pp. 321–326.

[18] ——, "Encoding of pointers for hardware synthesis," in *Proc. Int. Workshop IP-based Synthesis and System Design IWLAS'98*, Grenoble, France, Dec. 1998, pp. 57–63.

[19] L. Séméria, K. Sato, and G. 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, France, Mar. 2000, pp. 312–319.

[20] ——, "Memory representation and hardware synthesis of C code with pointers and complex data structures," in *Proc. Synthesis and System Integration of Mixed Technologies Workshop, SASIMI'00*, Kyoto, April 2000, pp. 43–48.

[21] L. Séméria and A. Ghosh, "Methodology for hardware/software co-verification in C/C++," in *Proc. Asia South Pacific Design Automation Conference ASP-DAC'00*, Yokohama, January 2000, pp. 405–408.

[22] B. Steensgaard, "Point-to analysis by type inference of programs with structures and unions," in *Proc. Int. Conf. Compiler Construction ICCC'96*, Apr. 1996, pp. 136–150.

[23] K. Wakabayashi, "C-based synthesis with behavioral synthesizer, cyber," in *Proc. Design, Automation and Test in Europe DATE'99*, Munich, Germany, 1999, pp. 390–391.

[24] P. Wilson, M. Johnstone, and D. Boles, "Dynamic storage allocation: A survey and critical review," in *Proc. Int. Workshop Memory Management*, Kinross, Scotland, Sept. 1995.

[25] R. Wilson, "Efficient, context-sensitive pointer analysis for C programs," Ph.D. dissertation, Stanford Univ., 1997.

[26] R. Wilson and M. Lam, "Efficient context-sensitive pointer analysis for C programs," in *Proc. ACM SIGPLAN'95 Conf. Programming Languages Design and Implementation*, June 1995, pp. 1–12.

[27] R. P. Wilson *et al.*, "Suif: An infrastructure for research on parallelizing and optimizing compilers," *ACM SIPLAN Notices*, vol. 28, no. 9, pp. 67–70, Sept. 1994.

[28] S. Wuytack, F. Catthoor, and H. De Man, "Transforming set data types to power optimal data structures," *IEEE Trans. Computer-Aided Design*, pp. 619–629, June 1996.

[29] S. Wuytack, J. da Silva Jr., F. Catthoor, G. de Jong, and C. Ykman, "Memory management for embedded network applications," *IEEE Trans. Computer-Aided Design*, vol. 18, pp. 533–544, May 1999.

[30] "C Level Design, C2HDL, http://www.cleveldesign.com/,".

[31] "CoWare, N2C, http://www.coware.com/,".

[32] "CynApps, http://www.cynapps.com/,".

[33] "Frontier Design, A/rt Builder, http://www.frontierd.com/,".

[34] "Synopsys CoCentric SystemC Compiler, http://www.synopsys.com/products/cocentric_systemC/cocentric_systemC.html,".

Luc Séméria received the Engineer degree from the Ecole Nationale Supérieure des Télécommunications, Paris, France, in 1996 and the M.S. and Ph.D. degrees in electrical engineering from Stanford University, Stanford, CA, in 1998 and 2001, respectively.

While studying he held several summer positions at Synopsys Inc. He is currently working on C/C++ design methodology at Clearwater Networks, Inc. His research interests include areas related to design, verification, and optimizing compilers.

Koichi Sato received the B.E. and M.E. degrees in electrical engineering from Waseda University, Japan, in 1990 and 1992, respectively.

In 1992, he joined NEC Corporation, where he is an Assistant Manager. In 1999, he was a Visiting Scholar in the Electrical Engineering Department, Stanford University, Stanford, CA. His research interests include system-level design, hardware/software codesign, timing optimization in logic synthesis, and layout-driven synthesis.

Giovanni De Micheli (S'79–M'82–SM'89–F'94) is a Professor of Electrical Engineering, and by courtesy, of Computer Science at Stanford University, Stanford, CA. His research interests include several aspects of design technologies for integrated circuits and systems, with particular emphasis on synthesis, system-level design, hardware/software codesign, and low-power design. He is the author of *Synthesis and Optimization of Digital Circuits* (New York: McGraw-Hill, 1994) and coauthor of four other books. He was also Co-Director of the NATO Advanced Study Institutes on Hardware/Software Co-Design, held in Tremezzo, Italy, 1995, and on Logic Synthesis and Silicon Compilation, held in L'Aquila, Italy, 1986.

Dr. De Micheli received a Presidential Young Investigator award in 1988. He received the 1987 IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN/ICAS Best Paper Award and two Best Paper Awards at the Design Automation Conference, in 1983 and in 1993. He received the IEEE/CAS Golden Jubilee Medal for his outstanding contribution to the IEEE/CAS Society in 2000. He was Vice President (for publications) of the IEEE CAS Society from 1999 to 2000. He is the Editor of the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN/ICAS. He was the Program Chair and General Chair of the Design Automation Conference (DAC) in 1996–1997 and 2000, respectively. He was also Program and General Chair of International Conference on Computer Design (ICCD) in 1988 and 1989, respectively.