Resolution, Optimization, and Encoding of Pointer Variables for the Behavioral Synthesis from C

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

***Abstract*—As designers may model mixed hardware–software systems using a subset of C or C++, we present SpC, a solution to synthesize and optimize hardware C models with pointers. In hardware, a pointer is not only the address of data in memory, but it may also reference data mapped to registers, ports, or wires. Pointer analysis is used to find the set of locations each pointer may reference in a program at compile time. In this paper, we address the problem of synthesizing and optimizing pointers to multiple variables or array elements. The value of the pointers are encoded and branching statements are used to dynamically access data referenced by pointers. A heuristic is used to efficiently encode the values of the pointers. Compiler techniques are also used to reduce storage before loads and stores. An implementation using the SUIF framework (Wilson *et al.*, 1994; SUIF Compiler Framework) is presented, followed by some case studies and experimental results.**

I. Introduction— Synthesis from C

DIFFERENT languages have been used as input to behavioral 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. C/C++ offers fast simulation as well as a vast amount of legacy code and libraries, which facilitate the task of system modeling. To implement parts of the design modeled in C/C++ in hardware using synthesis tools, designers must manually translate these parts into a synthesizable subset of HDL. This process is well known for being both time consuming and error prone.

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 hardware–software migration. Designers could describe their system using C/C++ and partition it into software and hardware blocks. Hardware synthesis tools from C/C++ would then be very useful to map C/C++ models into logic netlists.

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 [23]. This task turns out to be particularly difficult because of dynamic memory allocation, function calls, recursions, `goto`s, type castings, and pointers. The problem with dynamic memory allocation (`malloc, free`) and recursion is that the size of the memory required for an application is *a priori* unknown. Therefore, the synthesis of C code involving dynamic memory allocation would require access to an operating system running in software or the generation of hardware allocators [44], [53]. Arbitrary control flow (e.g., due to `goto` statements) complicates the scheduling of operations even though it has been addressed [49]. In general, the use of pointers is one of the major difficulties, especially when combined with pointer arithmetic and type casting. Pointers have different applications in C. They are often used in function calls to pass parameters by reference. They are also used to scan arrays, reference data structures, or perform any type of complex memory management operation. The semantic of pointers in C is the address of data in main memory. However, in hardware, designers may want to optimize the memory architecture by using registers, multiple memory banks, etc. Therefore, pointers cannot be considered as addresses to a single memory. To enable efficient mapping of C code with pointers to hardware, the synthesis tool has to automatically generate the appropriate circuit to access the data referenced by pointers. The resolution of pointers is a key feature for C-based synthesis. It is an enabler for fast data accesses and efficient scheduling of operations.

In this paper, we will focus on the efficient hardware implementation of pointers in C models. In Section II, we present some of the related work on synthesis from C as well as on compilation of C code onto parallel architectures. In Sections III and IV, we define our synthesizable subset of C and show how various types of pointers can be synthesized. In Sections V and VI, we discuss different techniques for optimizing the code by limiting the number of live variables before the loads and stores and encoding the value of the pointers. In Section VII, we present SpC, our framework for the synthesis and optimization of C code with pointers using the SUIF [52], [66] compiler framework and a commercial behavioral synthesis tool. Finally, in Section VIII, results are given for a set of examples.

II. Related Work

A. Hardware Synthesis from C/C++

Different subsets of C/C++ and C-like HDLs have been defined and used for synthesis. First, we mention those developed in the 1980s. HARDWAREC [26] is a language with a C-like syntax and a cycle-based semantic. It does not support pointers, recursion, or dynamic memory allocation. HARDWAREC can be fully synthesized. CONES [47] is an automated synthesis system that takes behavioral models written in a C-based language [6]

Manuscript received December 8, 1999; revised June 23, 2000. This work was supported in part by the ARPA, MARCO Gigascale Design Center, and by Synopsys Inc. This paper was recommended by Associate Editor R. Gupta.

The authors are with the Computer Systems Laboratory, Stanford University, Stanford, CA 95305 USA (e-mail: lucs@azur.stanford.edu; nanni@galileo.stanford.edu).

Publisher Item Identifier S 0278-0070(01)00942-3.

and produces gate-level implementations. Here, the Cmodel describes circuit behavior during each clock cycle of sequential logic. This subset is very restricted and does not 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 flows [12]. Constructs are added to model coarse-grain parallelism, communication, and data types. These constructs can be defined as new syntactic constructs, hence creating a new language. They can also be implemented as part of a $C{+}{+}$ class library [56], [69]. Even though restrictions on the language apply for synthesis, hardware–software systems are then modeled directly using $C{+}{+}$. Simulation is performed by running the executable, which is generated after compiling the models. Standard debugging environments can then be used to check the functionality of the system.

For reactivity, SYSTEMC [29], [69] (formerly known as SCENIC [28]) supports a mixed synchronous and asynchronous approach implemented as a $C{+}{+}$ class library. The Esterel C Language (ECL) [27] is synchronous because it is based on both C and ESTEREL. Other extensions include HANDEL-C [59] and BACH-C [22] (originally based on OCCAM), SPECC [65] (based on SPECCHART), CYNLIB [56], and C-LEVEL DESIGN [54].

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) that will eventually be synthesized using today's synthesis tools. The second approach consists of using $C/C{+}{+}$ directly as an input to behavioral synthesis.

In order to facilitate the mapping of C models into hardware, several tools exist that automatically translate C-based descriptions into HDL either at the behavioral level or the register transfer level (RTL) level. In the original BACHC compiler, a limited subset of C can be translated into VHDL at the behavioral level. COWARE [55], OCAPI [41], [62], CYNAPPS [56], and others [57], [70] automated the translation from a refined RTL model to HDL. These subsets do not include pointers.

Kim and Choi [24] as well as the authors of this paper [42], [44] were the first to report on the synthesis of hardware C models with pointers. Kim and Choi's implementation is limited to a rather small subset of C. Pointers that may point to multiple locations are not supported and such constructs as type casting and complex data structures are not considered. Two commercial tools, C-level design C2HDL [51] and frontier design AR|T BUILDER [58], also provide tools for translating C models into Verilog or VHDL. Limited scheduling and resource-sharing techniques can be applied to quickly generate RTL synthesizable code. Pointers are one of the limitations for AR|T BUILDER. Pointers are only supported to pass parameters by reference or to scan arrays (pointer arithmetic). These types of pointers can usually be removed using standard compiler techniques (propagation and function inlining) and by adding ports for procedures. C2HDL, on the other hand, supports all of the ANSI C constructs, excluding libraries. However, pointers are implemented in a software-like approach. They are only considered as addresses to data stored in memories that require the allocation of memories to store the various variables and addressing units. In hardware, designers may want to optimize the locality by storing data into multiple memories, registers, or even wires (e.g., output of functional units). Our tool SpC presented here enables such optimization by leveraging recent researches on pointer analysis and high-level synthesis.

Another approach is to use $C/C{+}{+}$ directly as an input to architectural synthesis tools. This approach has been chosen by Synopsys with COCENTRIC SYSTEMC COMPILER [19], [68] and by NEC with CYBER [49]. C and $C{+}{+}$ are both procedural imperative languages. Their semantics rely on an implicit Von Neuman architecture. The implementation of sequential functional descriptions into hardware has extensively been studied during the last decade [8], [17], [18], [25], [26], [60], [67]. Synthesis from $C/C{+}{+}$ description can leverage some of this previous work on architectural synthesis but also requires the development of some extensions for efficiently supporting the different constructs of $C/C{+}{+}$. Some of the current work on function calls as well as synthesis of structures in VHDL can also be relevant. More research is, however, required for supporting $C/C{+}{+}$ constructs such as pointers, dynamic memory allocation, and object-oriented features.

Finally, we should also mention some of the areas in which $C/C{+}{+}$ models mix hardware–software and other specific architectures. For hardware–software codesign, the COWARE N2C system [55] as well as its precursor [5] use $C/C{+}{+}$ as a language base for system specification. Additional constructs have been introduced to define concurrent processing blocks and communication. This description is used to synthesize the interfaces between the blocks. COSYMA [16] uses $C^{\star}$, another superset of C, with processes and timing constraints. During hardware synthesis, functions are inlined and pointers are treated only as memory references.

For synthesis of reconfigurable systems based on field programmable gate array (FPGA), several projects have been using $C/C{+}{+}$. For PAM-BLOX [33], a bottom-up methodology is presented in which a library of components can be defined and used as $C{+}{+}$ objects to build systems for the Pamette architecture. A similar design environment has also been developed based on C for SPLASH [20]. For mixed software and reprogrammable FPGA architectures, the GARP compiler [7] as well as the NIMBLE compiler [32] automatically generate retargetable coprocessors to speed up loops. Pointers are treated as references to the main memory. This approach is relevant for implementing memory-mapped input–output (I/O). However, it can be a limitation to parallelize data transfers inside of a datapath. Finally, Babb *et al.* [2] present a compiler for a variation of the RAW [71]parallel architecture in which one or multiple processing units can be replaced by specialized hardware blocks. The problem of pointers is addressed in order to map data to different memory tiles. pointers to multiple memory locations are, however, a limitation because these locations are mapped to a unique memory and, therefore, cannot be accessed in parallel in a datapath.

To summarize the previous work, pointers are one of the main outstanding issues for the synthesis of hardware from C. In order to guarantee good quality of results, the current practice is to support only a limited subset of the language with severe restrictions on pointers. Otherwise, a software-like approach is taken in which the data accessed by pointers are stored in memory. Our approach is based on the use of analysis techniques (*pointer analysis*) in order to generate efficient hardware from *C* code using any kind of pointers at the behavioral level.

B. Software Compilation of C and C++

C and *C++* are two of the most commonly used programming languages today. Many compilers exist for many different architectures. Most of the recent compilers not only try to map the different statements of the code into assembly instructions, but they also try to optimize the code for a given instruction set architecture (ISA). For distributed architectures, parallel compilers are trying to partition programs into multiple threads running in parallel. However, some of the *C* constructs such as pointers and arbitrary control-flow operations (`goto`, `longjmp`, etc.) make these optimizations difficult. In software, pointers represent addresses in memory. They are often used to pass parameters by reference, access array elements, address dynamically allocated memory, and manage the memory. By definition, pointers may reference multiple data, which happens when referencing the different elements of a data structure or an array. It may also happen inside of a function for pointers corresponding to parameters passed by reference or, more generally, when the value of the pointer at one point in the code varies according to the current context or the previous flow of operations.

Many of the optimizations done in today's compilers as well as in many high-level synthesis tools are based on data-flow analysis [1], [34]. The purpose of data-flow analysis is to provide information on how a code segment manipulates its data. Examples of applications include register allocation (based on reaching-definition and live-variable analysis), constant folding, common-subexpression elimination, loop optimization, dead-code elimination, etc. The optimizations presented in Section V are also applications of data-flow analysis. To solve a given data-flow problem, the effect of each programming language structure is modeled by transfer functions. The result of such transfer functions often depends on the data accesses at each statement in the program. Namely, in order to model the effect of statements involving a pointer, it is important to know what data may be accessed by the pointer (*points-to information*).

In order to parallelize programs onto distributed architectures, the independent sets of data, which can be processed in parallel, have to be extracted [30]. The problem here 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*. Different pointer-analysis techniques [37], [50], [51] exist. For hardware synthesis, we also need to know which variables are accessed at each statement. Therefore, pointer analysis can be used for the behavioral synthesis of *C* models as we will do in the next section.

III. Background—Synthesis of *C* Models with Pointers

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` declarations 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 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 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 to memory elements, registers, wires, or ports. They can be used to access 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.

The synthesis of hardware from *C* consists first of partitioning the memory. Each partition is then mapped to a 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 accessing 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 that the pointer may reference (i.e., dereference pointers).

Example 1: Consider the following code segment.

```
int *p, n;
int t[256];
struct { int a; int b; } in;
···
if (···)
  p = &in.a;
else
  p = &in.b;
···
t[n] = *p;
*p = t[n + 1];
···.
```

In the final implementation, we want to store array `t[]` in a memory and integer `n`, pointer `p`, and the two structure fields `in.a` and `in.b` in separate registers that are accessible in parallel. Moreover, pointer `p` may point to either `in.a` or `in.b`. If we associate the value `0` with `in.a` and `1` with `in.b`, we can remove the pointer. First, for the addresses (&), the assignments `p = &in.a` and `p = &in.b` can respectively be replaced by `p_tag = 0` and `p_tag = 1`, where `p_tag` represents the encoded value of the pointer. Second, the dereferences (⋆) in loads and stores can be removed as follows.

The load (`t[n] = *p`) can be replaced by the following:

```
if(p_tag == 0)
  t[n] = in.a;  /* case p == &in.a */
else
  t[n] = in.b;  /* case p == &in.b */.
```

The store (`*p = t[n+1]`) can be replaced by the following:

```
if(p_tag == 0)
  in.a = t[n+1];  /* case p == &in.a */
else
  in.b = t[n+1];  /* case p == &in.b */.
```

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 partitioned into a set of location sets as described in Section III-B. Subsequently, to synthesize load and store operations into hardware, we need to know at compile time the set of locations the pointers may reference (*points-to* information). As we have seen in Section II-B, such information is also widely used in compilers and can be determined by recent analysis techniques called *pointer analysis* or *alias analysis* described in Section III-C. Finally, in Section III-D, we present how memory can be partitioned into variables and arrays, which can be mapped to hardware.

A. Definition of the Subset

The ultimate goal of this research is to efficiently synthesize the full ANSI C. In this paper, however, we target mainly the synthesis of pointers to statically allocated data and explore different optimization techniques. Extensions of this work to include more of the C syntax (`malloc/free`) are possible [44], [45], but beyond the scope of this paper. 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 backend to our tool.

Our subset contains all statements supported by today's behavioral synthesis tools, including branches, loops, assignments, etc. It also contains pointers to data, which can be stored in multiple memories, registers, or wires. It supports pointers to statically allocated data such as variables, arrays and structures, pointers to pointers, and pointers to functions. Since memory blocks are instantiated at compile time, recursions and pointers to dynamically-allocated memory of which size is unknown at compile time are not allowed. This implies that in general, `malloc`, `free`, and recursions are not supported. Nevertheless, `malloc` followed by `free` could be allowed as well as tail recursion. Calls to `malloc` followed by `free` can be treated as local variables [44] and tail recursion elimination can be done by turning recursions into loops [34].

The pointer analysis techniques and the memory representation presented in the next sections support the complete ANSI C syntax. In this paper, however, we define our own synthesizable subset. Our subset includes all types of pointers and type casting. The code is assumed to be correct. Tools such as LCLint [63] or Purify [64] can be used to check that memory reads and writes are valid.

In addition, we set the following restrictions. One 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.

A second limitation stems from the fact that most commercial synthesis tools also 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. 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. The synthesis of functions in C and the resolution of pointers inside of functions are beyond the scope of this paper.

B. Memory Representation

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, which would distinguish each element of arrays or of recursive data structures, is not practical for large programs. As a result, most analysis techniques combine elements within a single data structure. In order to find both an accurate and practical representation for hardware synthesis, we use the notion of *location sets* introduced by Wilson and Lam [50], [51]. 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 $\{f + \text{is} | i \in \mathbb{Z}\}$ 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.

Example 2: In the code segment shown in Example 1, the memory can be represented by the following set of location sets: $\langle \texttt{p}, 0, 0\rangle$; $\langle \texttt{n}, 0, 0\rangle$; $\langle \texttt{t}, 0, 4\rangle$ for the elements of array `t`; $\langle \texttt{in}, 0, 0\rangle$ for `in.a`; and $\langle \texttt{in}, 4, 0\rangle$ for `in.b`. Note that offsets and strides are represented here as a number of bytes.

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. 1 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.

The representation of the memory itself depends on how locations are being accessed. Consequently, pointer analysis, which is the subject of the next section, and memory representation are tightly related.

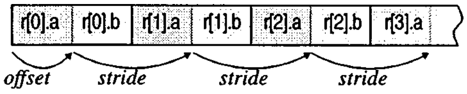


Fig. 1. Representation of `struct {int a; int b; } r[ ]`. The offset and stride are represented for the location set consisting of the elements `r[i].b` where `i` is an integer.

C. Pointer Analysis

Pointer analysis is a compiler pass to identify at compile time the potential values of the pointers in the program. This information is used to determine the set of locations that the pointer may point to. With the memory representation in Section III-B, this set of locations is actually a set of location sets. For synthesis in the case of loads and stores, we want to synthesize the logic to access or modify 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 less logic we have to generate. We can distinguish two types of analyzes.

1) *Flow and Context Insensitive*: This analysis [46] does not distinguish the order in which the statements are executed (*flow insensitivity*) and the different calls of a function (*context insensitivity*). This interprocedural analysis has an almost linear complexity. It can be used to analyze very large programs but the points-to information is rather inaccurate. Within a procedure, flow-insensitive analysis gives global information (valid for all references in the code) rather than the information specific to each reference. Similarly, in the case of function calls, context-insensitive analysis propagates the information from the call site through the called function and back to *all* call sites.
2) *Flow and Context Sensitive*: This analysis provides more accurate results. It distinguishes the different paths of control within the program and the different calls of a function. One implementation [50], [51] by Wilson and Lam within the SUIF framework can efficiently support the full-featured ANSI C with good accuracy. Even though the complexity of the analysis can be exponential, it is not a limitation for hardware synthesis because we deal with rather small and simple programs. In addition, most of the inaccuracy comes from features such as dynamic memory allocation, recursion, and recursive data structures that we do not consider in this paper.

The flow- and context-sensitive analysis is more appropriate for hardware synthesis. In our case, the complexity of the analysis is not an issue and the coding style for modeling hardware leads to accurate results.

Our implementation uses a flow- and context-sensitive analysis. Using the memory representation described in the previous section, the points-to information is defined as a set of location sets. The points-to information is then used to encode the pointers' value and to generate the appropriate logic for accessing the data in each location set.

Example 3: In the code segment presented in Example 1, annotations are inserted by the pointer analysis to specify where points-to set pointers may point at loads and stores.

```
int *p, n;
int t[256];
struct { int a; int b; } in;
...
if (···)
  p = &in.a;  //p→ {⟨in, 0,0⟩}
else
  p = &in.b;  //p→ {⟨in, 4,0⟩}
...
t[n] = *p;    //p→ {⟨in, 0,0⟩,⟨in, 4,0⟩}
*p = t[n+1];  //p→ {⟨in, 0,0⟩,⟨in, 4,0⟩}
···.
```

In the previous code segment, the notation $\langle\texttt{p, 0, 0}\rangle \rightarrow \{\langle\texttt{in, 0, 0 }\rangle,\langle\texttt{in, 4, 0 }\rangle\}$ stands for "p may point to variables `in.a` or `in.b`," where `in.a` is represented by the location set $\langle\texttt{in}, 0, 0\rangle$ and `in.b` is represented by the location set $\langle\texttt{in}, 4, 0\rangle$.

D. Memory Partitioning and Mapping to Variables and Arrays

After analysis, the storage in the program can be represented as a set of distinct location sets. This set of location sets represents a partitioning of the memory. Each partition block (i.e., each location set) is ultimately mapped to a wire, a register, or a section of memory in the final design. The allocation of a given variable to a register (or a wire) is typically the result of architectural synthesis. 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 [45], for each location set $\langle loc, f, s\rangle$, we define $\texttt{SPC}\_loc\_f\_s$ as the following.

For a singleton location set (i.e., s null), $\texttt{SPC}\_loc\_f\_s$ 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, the 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), $\texttt{SPC}\_loc\_f\_s$ is defined as an array (e.g., array of integers). Such array may then typically either be mapped to a memory or a register file manually or according to current methodology [9], [36]. 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 4: We have seen that `p, n, t, in.a,` and `in.b` in Example 1 can be represented by the location sets $\langle\texttt{p, 0, 0}\rangle$, $\langle\texttt{n, 0, 0}\rangle$, $\langle\texttt{in, 0, 4}\rangle$ for the elements of array t, ⟨in, 0, 0⟩ for in.a, and ⟨in, 4, 0⟩ for in.b. As a result, we create the following variables:

```
int SPC_n_0_0;   //n
int *SPC_p_0_0;  //p
int SPC_t_0_4[256];  //t[256]
int SPC_in_0_0;  //in.a
int SPC_in_4_0;  //in.b .
```

After partitioning, the code is then transformed into

```
...
if (...)
  SPC_p_0_0 = &SPC_in_0_0;
else
  SPC_p_0_0 = &SPC_in_4_0;
...
SPC_t_0_4[n] = *SPC_p_0_0;
*SPC_p_0_0 = SPC_t_0_4[n+1];
...
```

Note that the present code contains only variables and arrays. The partitioning process can be more complex with type casting and out of bound array accesses [45]. Nevertheless, after memory partitioning, the storage of the C program can be represented as a set of distinct variables and arrays. Therefore, in the remainder of this paper, all data are supposed to be either variables or arrays. In the next section, we present how pointers to variables and arrays can be synthesized. For clarity, variables and arrays such as SPC_a_0_0 and SPC_table_0_4[⋯] will be denoted a and table directly.

IV. Pointer Synthesis

In hardware, as discussed in Section III, 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. Pointer analysis is used to define the set of locations as a set of location sets that each pointer may point to. Our synthesis tool generates the appropriate circuit to dynamically access these locations according to the pointers' value. 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 (i.e., one location set whose stride is null) are simply replaced by assignments from the location accessed. Similarly, stores are simply replaced by assignments to the location referenced. Loads and stores from pointers to multiple locations (i.e., many location sets with zero strides and/or one or more location set with nonzero stride) are replaced by a set of assignments in which each location may be dynamically accessed according to the pointer's value. For the sake of clarity, we will use the variable name p as a generic pointer name.

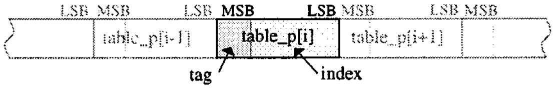


Fig. 2. Encoding of pointers in an array.

A. Encoding the Value of the Pointers

The addresses (i.e., pointers' values) are encoded. The encoded value of a pointer p consists of two fields: the *tag* p.tag corresponds to the location set referenced by the pointer and the *index* p.index stores the number of bytes corresponding to the offset of the data referenced within the location set.

The tag p.tag is only used for pointers to multiple location sets. Its size (defined as the minimum number of bit used to store its value) can be as small as ⌈log$_2$ (size_of_points-to-set)⌉. The index p.index, on the other hand, is used when the pointer p may point to a location with nonzero stride (e.g., an array). Pointer arithmetic is then supported by changing the value of the index: the value of p.index is initialized when p gets the address of the array element. Then the index is modified instead of p.

For pointer variables, the following two fields can be implemented as separate variables: p_tag and p_index.

Definition 1: For a pointer variable p we define the variables p_tag and p_index, where p_tag encodes the location set the pointer points to and p_index stores the offset corresponding to the location referenced by the pointer within the location set.

In the case of an array of pointers, the tag and index fields are merged into one data structure, as shown in Fig. 2. To support type casting, it is convenient to set the size of this data structure to be the same as the size of a pointer before encoding (typically 32 b). The tag is stored on the left part of the code and the index on the right part of the code to support pointer arithmetic.

Example 5: Consider the following code segment:

```
int *p, *q;
int a, b, c, table[256];
...
q = &table[n];
if (...) {
  p = &a;
  q = &c;
} else {
  p = &b;
  q = &table[n];
}
...
*q = *p + 1;
```

In this code segment, p may point to variables a or b and q may point to c or an element of table[ ]. In order to remove the pointers, we create the 1-b variables p_tag and q_tag. Since q may point to an array element, we also create the index q_index. For p_tag we associate the value 0 with a and 1 with b. As a result, the assignment p = &a is replaced by p_tag = 0 and the assignment p = &b is replaced by p_tag = 1. Similarly, for pointer q, we associate the value

0 with c and 1 with the location set representing the elements of table[ ]. The assignment q = &c is then simply replaced by q_tag = 0. The assignment q = &table[n] is replaced by two assignments: q_tag=1 and q_index=n*4.

Example 6: Consider the assignment of pointers (r = s), where r may point to a, b, or c and s may point to b or c. In order to remove the pointers, we create r_tag and s_tag. For r_tag, we associate the value 0 with a, 1 with b, and 2 with c. For s_tag, we associate 0 with b and 1 with c. The following code is generated for r = s:

```
switch s_tag:
  case 0: r_tag=1;
  case 1: r_tag=2;
```

Now, if for r_tag the value 0 was associated with b and the value 1 was associated with c, r = s would have been replaced by

```
r_tag = s_tag;
```

This shows that the complexity of the circuit implementing the assignment of two pointers is directly related to the encoding of the pointers. Efficient pointer comparison and assignment of pointers require pointers to have the same code or at least codes as close as possible.

The encoding of the pointers' value has an effect on the complexity of the design. Example 6 gives two examples of encodings that produce different implementations for the assignment of two pointers. In Section VI, the encoding problem is formulated and a heuristic solution is presented.

B. Dereferencing the Pointers

Several types of pointers can be distinguished. We have seen in Section III-D how complex data structures can be represented as variables and arrays. Without loss of generality, in this section we first consider pointers that may point to variables and array elements. We then present two extensions for pointers to pointers and pointers to function.

1) Pointers to Variables and Arrays: We use the result of pointer analysis to remove loads and stores. With the assumptions of Section III-A, loads and stores can be replaced by branching statements (e.g., case, if then else) at compile time. Pointer analysis defines the set of location sets that the pointer may reference at each load and store. When these location sets are mapped to registers or wires (e.g., output of a functional unit), the branching statements corresponding to a load are implemented using a multiplexer controlled by the pointer's value. In the case of a store, some control logic is generated to update the value of the variable the pointer points to. This control logic can be automatically generated by an architectural synthesis tool. References to array elements stored in memories or register files are treated similarly. Some control logic is also created to access the location referenced in the different memories or register files.

Example 7: Consider the code segment in Example 5: *q = *p+1, where p may point to a or b and q may point to either c or an element of table[ ]. To synthesize the load, we create the temporary variable star_p, which stores the value of the data the pointerp points to (i.e., *p) at the load instruction. Similarly for the store, we create the temporary variable tmp_q, which stores the new value to be assigned to the data q points to at the store instruction. After encoding the pointers' value (cf. Example 5), the loads and stores are then replaced by the following code:

```
switch p_tag: {
  case 0: star_p = a; break;
  case 1: star_p = b; break;
}
tmp_q = star_p + 1;
switch q_tag: {
  case 0: c = tmp_q; break;
  case 1: table[q_index]=tmp_q; break;
}.
```

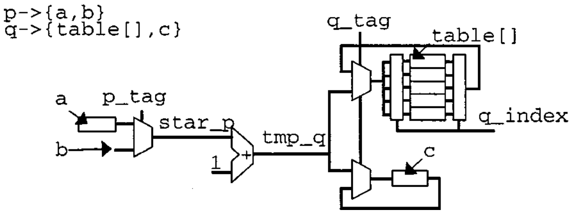


Fig. 3. Implementation of *q = *p+1, where p may point to a or b and q may point to c or an element of table[ ].

The corresponding circuit generated after synthesis is presented in Fig. 3. Note that the load ($\cdots =$*p) is implemented by a two-input multiplexer controlled by p_tag.

The removal of the dereferences "$\star$" in loads and stores can be done in one pass. For each load ($\cdots =$*p), we look at the points-to set of the pointer at this instruction. If the points-to set is only one location, the load is simply replaced by an assignment from this location. Otherwise, we create a temporary variable (star_p in Example 7) that stores the value of the data the pointer points to at the load instruction. The load instruction is then replaced by an assignment from this temporary variable. Branching statements are inserted before the load to set the value of the temporary variable star_p according to the values of the tag p_tag and the index p_index.

Similarly, for each store (*q = $\cdots$), we also look at the points-to set of the pointer q at this instruction. If the pointer points to only one location, the store is simply replaced by a assignment to this location. Otherwise, we create a temporary variable (tmp_q in Example 7) that stores the value to be assigned to the data q points to. The store is then replaced by an assignment to this temporary variable and branching statements are inserted after the store to update the values of the variables that q may point to according to the tag q_tag and index q_index.

This implementation can be generalized to pointers to pointers and pointers to functions. In Section V, we also present some optimizations to reduce the memory usage before loads and between loads and stores when the pointer is a variable.

2) Generalization to Other Types of Pointers: In general, pointers may also point to other pointers and functions. The technique presented in the previous section can be extended to these types of pointers.

Pointer to Pointers: Pointers to pointers can be implemented by resolving the pointers level by level.

Example 8: Consider a pointer p that may point to two pointers q1, q2. Pointers q1 and q2 may in turn both point to variables a or b. The statement (**p = **p+1;) can be resolved as the following by using a sequence of two case statements. For the sake of clarity, the pointer's values have not been encoded. Encoding of the pointer's value can be performed in a second pass.

```
switch p {
  case &q1:
        star_p = q1; break;
  case &q2:
        star_p = q2; break;
}
switch star_p {
  case &a:
        star_star_p = a; break;
  case &b:
        star_star_p = b; break;
}
tmp_star_p = star_star_p + 1;
switch p {   //Note: can be removed by
                     further analysis
  case &q1:         //
        star_p = q1;  //
        break;        //
  case &q2:         //
        star_p = q2;  //
        break;        //
}                   //
switch star_p {
  case &a:
    a = tmp_star_p; break;
  case &b:
    b = tmp_star_p; break;
}.
```

A better implementation can be obtained by removing unnecessary definitions. In the previous example, the third switch statement redefining star_p is not necessary and can be automatically removed using compiler analysis techniques.

Pointer to Functions: Pointers to functions are resolved in a straightforward manner after pointer analysis.

Example 9: For a pointer p that may point to functions f1(int) *or* f2(int), (*p) (a) will simply be replaced by the following code segment:

```
switch p {
  case &f1: f1(a); break;
  case &f2: f2(a); break;  }.
```

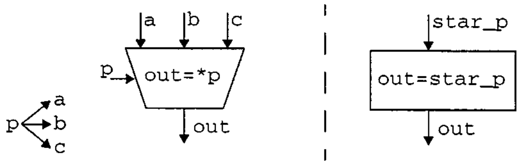


Fig. 4. Optimization of a load.

In order to map this code into hardware, functions f1 and f2 can be inlined and the value of pointer p is encoded.

The synthesis of the functions themselves is then performed according to the synthesis tool (e.g., map to component, inline ...). In our implementation, functions are inlined before synthesis.

V. Optimization of Loads and Stores

In the previous section, we have seen how pointers can be removed after pointer analysis. Now we optimize the code for hardware synthesis. First, we present techniques to reduce the amount of storage necessary before loads (··· =*p) and stores (*p = ···) when the pointer p is a variable.

In this section, the following assumptions are made. The pointer p is a variable. Its points-to set consists of a set of variables (mapped to registers or wires). The optimizations presented here are only performed when the previous assumptions hold. Their generalization to loads and stores from pointers within an array or pointers pointing to array elements is beyond the scope of this paper.

The goal of the optimizations presented here is to reduce the number of live variables[1] before loads and stores. When variables are stored in registers, the number of registers used in a given program corresponds to the maximum number of variables live at a clock boundary. The direct effect of our optimizations is, therefore, to reduce the number of registers used in the design. Besides, synthesis tools may also take advantage of having less live variables before loads and stores to improve performance by more efficiently reusing registers.

A. Optimization of Loads

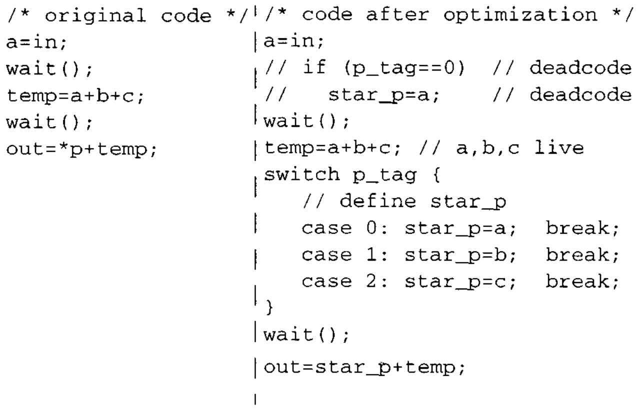
By definition, a load may read any variable of the points-to set. It also uses the value of the pointer to select which variable is actually read. This implies that all variables of the points-to set and the pointer variable are live before the load. However, only one variable is really necessary: the variable the pointer points to.

Definition 2: For a pointer variable p, we define star_p as a variable whose value is equal to the value of the data the pointer p points to at any point in the program.

A load (··· =*p) is then equivalent to an assignment from star_p. The number of live variables before a load can then be reduced by at most the number of variables in the points-to set as we can see in Example 10.

Example 10: In Fig. 4, the load (out = *p), where p may point to a, b, or c, is replaced by an assignment from star_p.

[1] A variable is *live* at a particular point in a program if there is a path to the exit along which its value may be used before it is redefined (i.e., *killed*). It is *dead* if there is no such a path [1], [34].



```
/* original code */| /* code after optimization */
a=in;              | a=in;
wait();            | // if (p_tag==0)  // deadcode
temp=a+b+c;        | //   star_p=a;    // deadcode
wait();            | wait();
out=*p+temp;       | temp=a+b+c; // a,b,c live
                   | switch p_tag {
                   |    // define star_p
                   |    case 0: star_p=a;  break;
                   |    case 1: star_p=b;  break;
                   |    case 2: star_p=c;  break;
                   | }
                   | wait();
                   | out=star_p+temp;
```

Fig. 5. Example of code segment before and after optimizing load.

The number of live variables before the load goes from 4{a, b, c, p} to 1{star_p}, assuming that none of these variables are live after the load.

The issue is then to define star_p in such a way that the number of live variables is reduced. In our implementation, each load is replaced by assignments from star_p. The variable star_p itself is defined each time p or any variable in the points-to set is modified. Dead-code elimination [1], [34] is then performed to remove all unnecessary definitions of star_p.

However, the early definition of star_p may also increase the number of live variables. When all variables of the points-to set are live, star_p is just a copy of one of these variables and is not necessary. Therefore, in order to minimize the number of live variables, star_p is *killed* (i.e., redefined) when all variables of the points-to set are live. The following is an outline of the complete algorithm for the optimization of loads.

1) Update star_p when p or any variable of the points-to set changes.
2) Do live variable analysis [1], [34] (implemented as backward data-flow analysis).
3) Insert definition of star_p when all variables of the points-to set are live.
4) Do dead-code elimination.

Example 11: Let us take the code segment shown in Fig. 5 before and after optimization, where the pointer p may point to a, b, or c.

We assume that none of the variables are live after the last line. During the first pass, we replace *p by star_p and update star_p after a = in. Then, because of temp = a + b + c, a, b, and c are live at the first wait( ) statement. After live variable analysis, we add the case statements which define (i.e., kill) star_p. Finally, dead-code elimination will remove the first definition of star_p at the beginning of the code. The number of live variables before the load has been reduced from 5{a, b, c, p, temp} to 2{star_p, temp}.

This optimization can drastically decrease the number of live variables before loads. Nevertheless, it increases the number of branching statements, which correspond to combinational steering logic to control the value of star_p. Therefore, there is a tradeoff here between the number of live variables (i.e., registers) and the amount of steering logic in the hardware implementation.

B. Optimization of Stores

In this section, we try to apply the same idea of creating temporary variables to reduce the number of life variables before stores.

Example 12: Let p be a pointer that may point to a, b, or c. Consider the store *p = in, assuming that all variables of the points-to set are live after the store. As a result, we have five variables {p, in, a, b, c} live before the store. Now assume that, at runtime, p points to a. Since the value of a is going to be redefined by the store there, it is not needed before the store. As a result, the number of live variables before the store could be reduced by one. Note that the same applies when p points to b or c.

As we have seen in Example 12, the number of live variables before a store can be reduced by at most one. The reason is that the store needs all variables of the points-to set (that are live after the store) except the variable that p points to. For this purpose, given a pointer p and the size of its points-to set pts_size, we define the following class of variables:

```
_starN_p, for N in {1, 2,⋯,
  (pts_size-1)}.
```

("_starN_p" stands for "not star_p"), variables whose values are equal to the values of the variables in the points-to set p, does *not* point to.

Note that each _starN_p can be defined in such a way that it may only store the value of either variables of a fixed pair as shown in Example 13.

Example 13: If p may point to a, b, or c, we create _star1_p and _star2_p and define them as the following (note that other formulations may be used):

```
_star1_p = (p!=&a)?a:b;
_star2_p = (p!=&b)?b:c;
```

As a result, the store *p = in, which leads to five live variables (cf. Example 12) can be replaced by the following code segment, which uses only four variables {p, _star1_p, _star2_p, in}:

```
switch p: {
  case &a:  a = in; b = _star1_p;
            c = _star2_p; break;
  case &b:  a = _star1_p; b = in;
            c = _star2_p; break
  case &c:  a = _star1_p; b = _star2_p;
            c = in; break;
}.
```

To optimize the number of live variables before stores, let us first consider an adaptation of the algorithm described in Section V-A. Indeed, one could imagine an algorithm where the _starN_p variables are used at each store and defined when p or any variable of the points-to set is modified. Since each _starN_p variable can only store the value of one of two variables of the points-to set, they should be *killed* each time one of the variables of the points-to set is live. For hardware synthesis, this creates a lot of logic to control their value, which turns out not to be very practical.

In our implementation, we take a conservative approach by optimizing stores only in the case of a load followed by a store. Such a case happens after inlining functions in which the parameters passed by reference are both read and written within the function.

Example 14: Let us look at the example of (`*p = *p+1`), where `p` may point to `a` or `b`. Such a code may be generated after inlining the function call `incr(p)`, where `incr(int *)` is defined as

```
incr(int*q){*q = *q + 1;}.
```

The code corresponding to (`*p = *p+1`) after optimization using `_star1_p` is the following:

```
// definition of star_p and _star1_p
if(p == 0) {
  star_p = a;
  _star1_p = b; }
else {
  star_p = b;
  _star1_p = a; }

star_p = star_p + 1;

// assignements to a and b
if(p == 0) {
  a = star_p;
  b = _star1_p; }
else
  b = star_p;
  a = _star1_p; }.
```

Fig. 6 shows the control data-flow graph (CDFG) before and after optimization. The definition of the temporary variables has been inserted before the load and the variables of the points-to set are updated after the store. We can verify that the number of live variables between the load and store has been reduced from four {`a, b, p, star_p`} to three {`star_p, _star1_p, p`}.

For a pointer `p`, the algorithm for reducing the number of live variables between loads and stores is the following.

1) List the stores dominated[2] by loads from the same pointer (implemented as a forward data-flow analysis [1], [34]).
2) List the loads postdominated[3] by stores from the same pointer (implemented as a backward data-flow analysis [1], [34]).
3) Do live variable analysis assuming that each store in the list generated at Step 1 kills all variables in the points-to set.

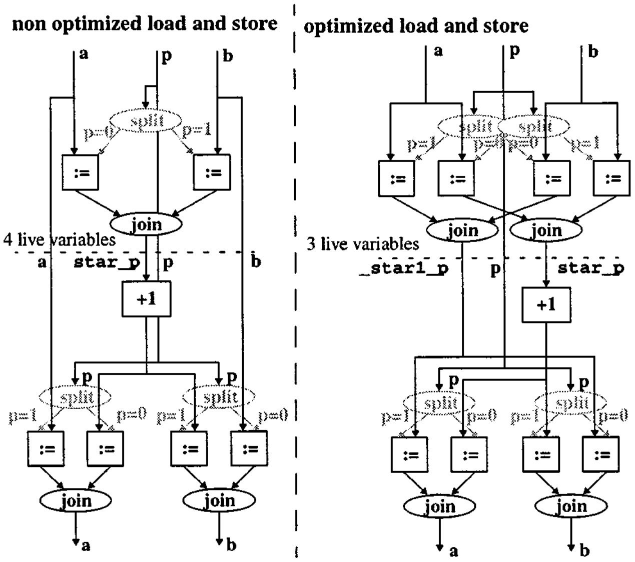


Fig. 6. CDFG for `*p = *p+1` with `p`→{`a, b`}.

4) If for all loads in the list generated at Step 2 none of the variables in the points-to set are live:
   - define `star_p` and the `_starN_p` variables before the loads and when `p` or any variable of the points-to set changes between loads and stores;
   - use `star_p` and the `_starN_p` variables to update the values of variables in the points-to set after the stores.

Even though this optimization reduces the number of live variables before stores by at most one, it helps reduce the number of registers. There is, however, a tradeoff between the number of registers used and the amount of steering logic. This optimization can be performed while optimizing the loads, as we will see in Section VII.

VI. Encoding of Pointers

In software, the pointers' values represent addresses in memory. These values are used in loads and stores; they have a fixed size and can then be assigned (`p = q`) or compared (`p == q`). In hardware, we want to reduce the size of the storage and the complexity of the decoding logic in loads and stores. In Section IV-A, we have seen that the encoding of a pointer consists of two field: a tag and an index. In this section, we are trying to encode the tag part more efficiently. Other techniques similar to the encoding of memory addresses [4], [36] could be used to encode the index part, although they are not addressed in this paper.

Definition 3: We define the size of a pointer as the bit width of its tag.

When the size of the pointer is decreased, the number of bit registers used to store its value is also reduced. The decoding logic for loads and stores is also simplified. We have seen that a load can be implemented as a multiplexer controlled by the pointers' value (tag part). Reducing the pointers' size simplifies also the complexity of the decoding logic for this multiplexer. However, as we have seen in Example 6, when pointers are assigned or compared, we may have to add `case` statements to

[2] Instruction d dominates instruction i in a flowgraph if every possible execution path from the entry node to i includes d [34].

[3] Instructions i post-dominates instruction d in a flowgraph if every possible execution path from the d to the *exit* (aka *sink*) node includes i [34].

"translate" the values of the pointers by means of some combinational circuit. We can use encoding techniques to minimize the size of these circuits. Our goal is twofold: 1) we want to encode each pointer with the minimum number of bits in order to minimize the storage as well as the decoding logic for loads and stores; and 2) we want to minimize the logic related to assignment and comparison of pointers.

We will first present the problem of pointers' encoding. The exact solution to this problem leads to what we call a *local encoding* in which two pointers that point to the same location set may have different encodings. This problem is, however, hard to solve and a heuristic is then introduced in which two pointers that point to the same location set share the same encoding. This gives a *global encoding* of the pointers' value. In order to get closer to the exact solution corresponding to the local encoding, two optimizations are then presented called *splitting* and *folding*. These optimizations can be seen as adding "locality" to the global encoding.

A. Definition of the Problem

In this section, we present the problem of encoding the value of the pointers. Our first goal is to minimize the size of the pointers. Then, when a pointer is assigned or compared to another pointer, we want the corresponding tags to be equal (e.g., `p_tag = q_tag`) or "as close as possible" to each other. If two tags have different bit width, one tag can be equal to a subfield of the other. Assignments would then be performed by concatenating or removing bits, whereas comparisons would only be executed on subfields of the two codes. This reduces the size of the circuit that translates or compares the tags while keeping the number of bits to a minimum.

Definition 4: For two pointers p_i and p_j, the pointer dependence relation $r(p_i, p_j)$ is one if and only if the two pointers are assigned or compared (otherwise it is zero).

Definition 5: The *pointer-dependence graph* is an undirected graph in which the nodes are the pointers and the edges are the relations between the pointers. An edge between two nodes is defined when the two corresponding pointers are assigned or compared.

Example 15: Consider the following code segment:

```
int *r1, *r2, *r3, *q1, *q2;
...
if(i == 0)
  { r1 = &a; r2 = &b; r3 = &c; }
else
  { r1 = &b; r2 = &c; r3 = &d; }

if(j == 0)
  { q1 = r1; q2 = r2; }
else
  { q1 = r2; q2 = r3; }
....
```

In this example, we consider the pointers {`r1, r2, r3, q1, q2`} and the variables {`a, b, c, d`}. The pointers are defined as follows: `r1` may point to the variables `a` or `b`, `r2` may point to `b` or `c`, and `r3` may point to `c` or `d`. Then, `q1` may take the value of `r1` or `r2` and `q2` may take the value of `r2` or `r3`. Consequently, `q1` may point to `a`, `b`, or `c` and `q2` may point to `b`, `c`, or `d`.

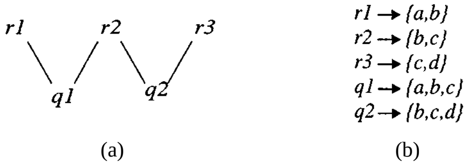


Fig. 7. (a) Example of pointer-dependence graph. (b) Definitions of the points-to sets of each pointer.

This leads to the pointer-dependence graph in Fig. 7(a).

The encoding problem can be stated as follows. For each pointer we represent its points-to set as a set of symbols corresponding to the location sets the pointer may point to. Thus, we have an ensemble of sets of symbols and the dependencies among the sets represented by the *pointer-dependence* graph. The problem consists of encoding the symbols in the sets. There are two constraints on the encoding: 1) the supercube[4] of the codes of the symbols in each set must have minimum size and 2) the symbols that correspond to the same location set in two dependent sets must be encoded as close as possible. The reasons for the first constraint are to minimize the number of bits to store and to reduce the decoding logic for loads and stores. The reason for the second constraint is to reduce the size of the combinational circuit implementing pointers' assignments and comparisons.

Example 16: In Example 15, the pointers `r1`, `r2`, and `r3` may point to two different variables and `q1` and `q2` may point to three different variables. As a result, we want to encode pointers `r1`, `r2`, and `r3` on 1 b and pointers `q1` and `q2` on 2 b.

Fig. 8(a) shows an example of a nonoptimal encoding. The encoding technique used here is a straightforward minimum-length encoding in which the value "0" is assigned to the first variable in the points-to set, 1 is assigned to the second variable of the points-to set, etc. This encoding is not optimal; some logic has to be added in the circuit to implement the assignments `q2 = r3` and `q1 = r2`, as shown in Fig. 8(a).

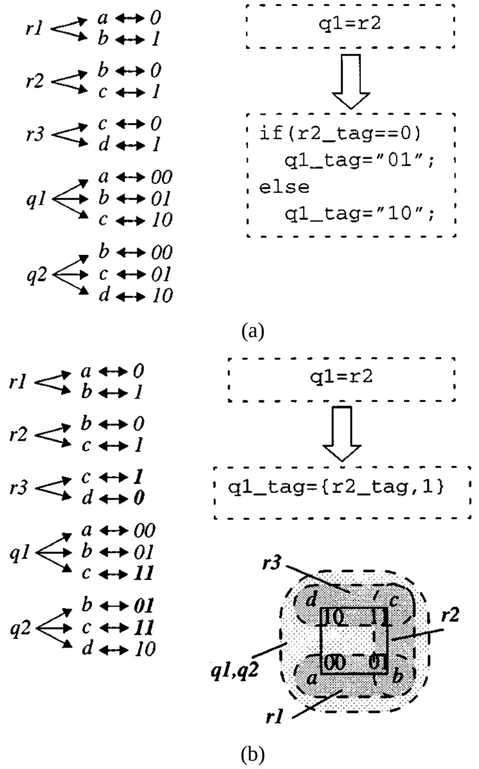
To find an optimal encoding, we look at the dependence between the pointers. Pointer `q1` may take the value of `r1` or `r2`. Therefore, we want the codes of `r1` and `r2` to be subfields of the code of `q1`. Similarly, `q2` may take the value of `r2` or `r3`. We want the codes of `r2` and `r3` to be subfields of the code of `q2`. An optimal encoding verifying these properties is shown on Fig. 8(b).

For `r1`, value 0 is assigned to `a` and value 1 to `b`. For `r2`, 0 will be assigned to `b` and 1 to `c`. As a result, `q1 = r1` will be replaced by `q1_tag = {0, r1_tag}` and `q1 = r2` will be replaced by `q1_tag = {r2_tag, 1}` (where `{,}` is the concatenation operator).

B. Problem Formulation

Let us consider P pointers $\mathrm{P} = \{\mathrm{p}_1, \mathrm{p}_2, \ldots, \mathrm{p}_\mathrm{P}\}$. For each pointer $p_i \in \mathrm{P}$, let Π_i be its points-to set. The points-to set Π_i

[4] The *supercube* of a set of cubes is the smallest cube containing all the cubes in the set [10].



```
q1=r2
```

```
if(r2_tag==0)
  q1_tag="01";
else
  q1_tag="10";
```

(a)

```
q1=r2
```

```
q1_tag={r2_tag,1}
```

(b)

Fig. 8. Example of (a) nonoptimal and (b) optimal encoding. Codes that are changed in the optimal encoding are shown in bold.

is a set of N_i symbols $\Pi_i = \{s_1^i, s_2^i, \ldots, s_{N_i}^i\}$, where each symbol is associated with a location set. We define E_i the set of the encoded symbols of the points-to set Π_i. The encoded values of the symbols in each set are noted $\{e_1^i, e_2^i, \ldots, e_{N_i}^i\}$.

Definition 6: Two sets Π_i and Π_j are said to be dependent if their associated pointers are dependent (Definition 4).

Our first goal is to minimize the number of bit registers as well as the size of the decoders required to store and decode the pointers' values. We want to minimize the dimension of the supercube of the encoded symbols in each set. This minimum is achieved when the sum of the dimensions of supercubes is also minimized

$$\min\left(\sum_{i=1}^{P} \mathrm{dim}(\mathrm{supercube}(E_i))\right). \tag{1}$$

Example 17: In the encoding presented in Fig. 8(a) and (b), $\sum_{i=1}^{P} \mathrm{dim}(\mathrm{supercube}(\mathrm{E_i})) = 1 + 1 + 1 + 2 + 2 = 7$ is minimum.

When two pointers are assigned or compared, we also want to minimize the size of the circuit implementing the translation of the codes. For this purpose, the distance between encoded symbols in two dependent sets has to be minimum

$$\min\left(\sum_{i=1}^{P}\sum_{j=1}^{P} r(p_i, p_j)\mathrm{dist}(E_i, E_j)\right) \tag{2}$$

where $\mathrm{dist}()$ is the distance between the two encoded sets. When the pointers have the same points-to set and the encoding has the same length n, $\mathrm{dist}()$ is defined as

$$\mathrm{dist}(E_i, E_j) = \min_{\mathrm{perm}()}\left(\sum_{k=1}^{N} H(\mathrm{perm}(\mathrm{e_k^i}), \mathrm{e_k^j})\right) \tag{3}$$

where

$N = N_i = N_j$ number of symbols in the points-to sets;
$\mathrm{perm}()$ set of the permutation functions of n bits;
$H(a, b)$ Hamming distance.

Note that the two equal points-to sets may have different encodings.

In general, the points-to sets may differ and their encoding may have different lengths. The computation of the distance is then more complex. For example, the distance between two sets whose encodings have different lengths can be computed by padding the shorter codes with zeros or ones. Then, if the points-to sets Π_i and Π_j differ, we are only interested in the distance between the encoding of the symbols common to the two points-to sets.

Our goal is to minimize (1) and (2). There is a tradeoff between the storage area (number of registers) and the amount of logic used to translate the codes. For example, one may optimize the size of the pointers keeping the amount of logic minimum by minimizing first (2) and then (1). In general, we can cast the problem as the following:

$$\min\left(\beta\sum_{i=1}^{P} \mathrm{dim}(\mathrm{supercube}(E_i)) + (1-\beta)\sum_{i=1}^{P}\sum_{j=1}^{P} r(p_i, p_j)\mathrm{dist}(E_i, E_j)\right) \tag{4}$$

where β is a coefficient between zero and one.

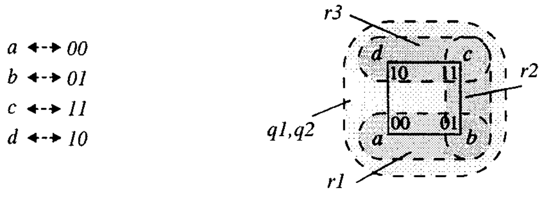
Since this problem is computationally hard to solve, we use heuristics.

C. Simplified Problem

1) Formalism for a Global Solution: In the general formulation of the problem presented in Section VI-B, different codes may be associated with the symbols in each set. Therefore, the encoding has to be found *locally* for each set. The problem can be simplified by constraining all symbols associated with the same location set to share the same code. The encoding is then found *globally* for all the symbols that correspond to the same location set in the points-to sets. The final encoding values of the pointers is then found by picking the relevant bits (i.e., the bits that are not identical for the different encodings of the symbols in the points-to set).

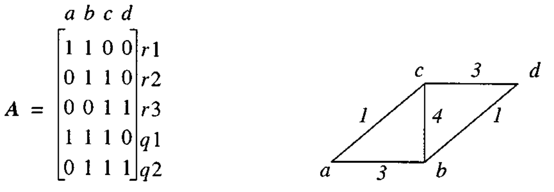
Example 18: Fig. 8(a) gives an example of local encoding. It is a local encoding because the different variables `a`, `b`, `c`, and `d` are associated with different codes in each points-to set. For example, `b` is associated with `1` for `r1` and `0` for `r2`.

Fig. 8(b) gives an example of a better global encoding. The encoding is global because the pointers initially share the same encoding shown in Fig. 9. No circuit is necessary to translate the values of the pointers in assignments and comparisons. The size of each pointer can be reduced by selecting the relevant bits for each pointer. These relevant bits are found as the following. Pointer `r2` may point to `b` or `c`. In the global encoding, value `01` is assigned to `b` and `11` is assigned to `c`. The value of the second bit in the encoding is then constant equal to `1` for the two encoded symbols in the points-to set of `r2`. As a result, pointer `r2` does not need to store this bit and the size of `r2` can be



(a) (b)

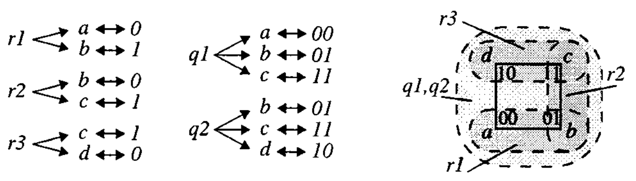
Fig. 9. (a) Global encoding. (b) Selection of the relevant bits for each pointer.



$$A = \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \left[\begin{array}{cccc} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \end{array}\right] \end{array} \begin{array}{c} \\ r1 \\ r2 \\ r3 \\ q1 \\ q2 \end{array}$$

(a) (b)

Fig. 10. (a) Example of relation matrix. (b) Corresponding affinity graph.



(a) (b)

Fig. 11. (a) Example of optimal encoding. (b) Corresponding representation in the Boolean hyperspace.

reduced to 1 b. Similarly, the size of `r1` and `r3` can be also be reduced to 1 b.

For a global encoding, minimizing (2) is then irrelevant because the distance between the codes of the symbols that correspond to the same location set in the different points-to sets is null (i.e., $\mathrm{dist}(E_i, E_j) = 0) \, \forall (i, j) \in \{1, 2, \ldots, P\}^2)$. The complexity of the logic to perform assignments and, to some extent, comparison is then minimal. However, the size of the pointers may vary and affect the size of the decoding circuit in loads and stores. Our goal becomes to minimize (1) only.

For this simplified problem, it is convenient to consider the symbols (i.e., location sets) in the union Π of the points-to sets. These symbols will be denoted: $\Pi = \{s_1, s_2, \ldots, s_N\}$. The size of the problem is reduced. Instead of dealing with $O(P \star N)$ symbols, we only deal with N symbols $\{s_1, s_2, \ldots, s_N\}$, where N is the number of location sets. We use now a formalism that has been used to solve other encoding problems [11], [48].

Definition 7: The relation matrix $\boldsymbol{A}$ is defined as the matrix in which the rows represent the points-to sets and the columns represent the symbols. Entry $A_{i,j}$ of $\boldsymbol{A}$ is one if and only if the symbol s_j is in the set Π_i.

Example 19: Let us take the case of Example 15, where `r1` may point to the variables `a` or `b`, `r2` may point to `b` or `c`, and `r3` may point to `c` or `d`, etc. We can construct the following relation matrix:

$$\boldsymbol{A} = \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \end{bmatrix} \end{array} \begin{array}{c} \\ r1 \\ r2 \\ r3 \\ q1 \\ q2 \end{array}.$$

For example, the first row of the matrix shows that `r1` may point to `a` or `b`.

We search for an encoding matrix $\boldsymbol{E}$. Namely, each row in $\boldsymbol{A}$ corresponds to a points-to set. For each row α of $\boldsymbol{A}$, we want the supercubes of the rows of $\boldsymbol{E}$ corresponding to the ones in α to have minimum size. This correspondsto the constraint expressed in (1). This problem corresponds to the input encoding problem [10], [11], [48] if the zeros in matrix $\boldsymbol{A}$ are replaced by *do not cares* (i.e., $*$). In other words, our problem is a simpler instance of the general input encoding problem.

2) Global Encoding Algorithm: The problem of input encoding has been extensively studied [3], [11], [15], [35], [38]–[40], [48]. We use an approach reminiscent of POW3 [3] and MUSTANG [35].

Definition 8: An affinity graph is an undirected weighted graph in which the nodes are the symbols $\Pi = \{s_1, s_2, \ldots, s_N\}$ and the edges are the relations between the symbols in Π, represented by the relation matrix $\boldsymbol{A}$. The weight $w_{i,j}$ on the edge $\{s_i, s_j\}$ is defined as the following:

$$w_{i,j} = \sum_{k=1}^{P} a_{k,i} \cdot a_{k,j} \cdot (1 + \lceil \mathrm{Log}_2 N \rceil - \lceil \mathrm{Log}_2 N_k \rceil) \quad (5)$$

where

P	number of pointers;
N	total number of symbols;
N_k	number of symbols in the set Π_k;
$a_{i,j}$	element of the relation matrix.

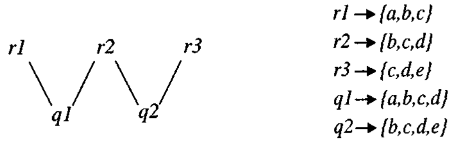
The weight $w_{i,j}$ in the affinity graph increases with the number of sets that contain both s_i and s_j. When two location sets are in many points-to sets, we want their codes to be close. This is even more important for small points-to sets. For example, if we have $N_k = 2$ symbols in the points-to set Π_k, their codes must be next to each other to minimize the dimension of the supercube of the encoded set E_k. Whereas if we have $N_k = 10$ symbols in the points-to set Π_k, the Hamming distance between the encoding of the symbols in the points-to set can be as much as $\lceil \mathrm{Log_2(N_k)} \rceil = 4$. Therefore, the weight $w_{i,j}$ is the sum of the contributions of the points-to sets that contain both s_i and s_j, where the contribution of each points-to set Π_k is $(1 + \lceil \mathrm{Log_2 N} \rceil - \lceil \mathrm{Log_2 N_k} \rceil)$.

The pointer encoding problem can be solved as an embedding of the affinity graph in the Boolean hypercube as done in [3], [21], [35], and [38].

Example 20: The relation matrix presented in Example 19 [cf. Fig. 10(a)] can be used to generate the affinity graph in Fig. 10(b).

Let us look at $w_{a,b}$, the weight on the edge `{a, b}`. The variables `a` and `b` are both in the points-to sets of `r1` and `q1`. The weight $w_{a,b}$ is 3, sum of 2, contribution from `r1`, and 1, contribution from q1.

After graph embedding, the encoding presented in Fig. 11 can be found. The graph embedding will try to put the encoding of the symbols that are adjacent to the edge of higher weight next to each other. As a result, the encoding of `b` is next to the encoding of `c` (edge `{b, c}` has a weight of four). The encodings of symbols `a` and `b` are also next to each other and so are the encodings of `c` and `d`.



(a) (b)

Fig. 12. (a) Pointer-dependence graph. (b) Definitions of the points-to sets.

Note that the aforementioned algorithms are solving a simplified problem (global encoding) in which all points-to sets share the same encoding. In order to better approximate the exact solution, two optimizations are presented in the next sections. In the exact solution (local encoding), two symbols can share the same code. We use this property in Section VI-D in a technique called *folding*. One symbol can also have multiple codes. The notion of *splitting* presented in Section VI-E is based on this property.

D. Encoding with Folding

In the local encoding problem, two symbols can share the same the code.

Definition 9: We define as folding the action of assigning the same code to two different symbols.

Proposition 1: Two symbols can be folded if and only if they are not both in the same points-to set and not in any two dependent points-to sets.

The rationale for this proposition is that we want to distinguish each symbol inside a points-to set and, in the case of a comparison, we want to distinguish the symbols in the two dependent points-to sets.

In the relation matrix $\boldsymbol{A}$, folding the symbols s_i and s_j are equivalent to replacing columns i and j by one column k such that

$$a_{k,l} = a_{i,l} \vee a_{j,l} \qquad \text{for } l \text{ in } \{1, 2, \ldots, N\}. \tag{6}$$

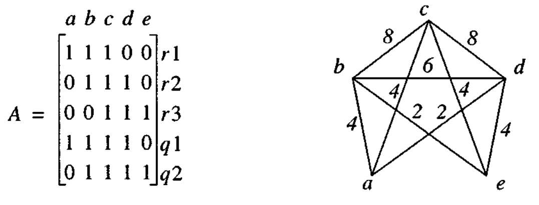
In the affinity graph, folding is done by merging (or fusing[5]) the nodes corresponding to the symbols s_i, s_j into one new node corresponding to s_k. The weights on the edges incident to this new node corresponding to s_k are then defined as

$$w_{k,l} = w_{i,l} + w_{j,l} \qquad \text{for } l \text{ in } \{1, 2, \ldots, N\}. \tag{7}$$

Graph-embedding techniques can be modified to incorporate folding. In Section VI-F, we present a column-based encoding algorithm with folding.

Example 21: Let us consider the pointer-dependence graph in Fig. 12, where r1, r2, and r3 point respectively to {a, b, c}, {b, c, d}, and {c, d, e}.

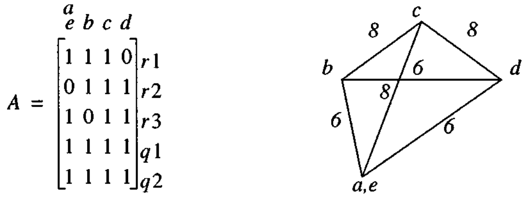
The relation matrix and the associated affinity graph are represented in Fig. 13. The number of variables (i.e., location set) in each points-to set is either three (for r1, r2, and r3) or four (for q1 and q2). Therefore, we want to code the symbols associated with the variables on 2 b. However, since we have five symbols, an encoding with less than 3 b cannot be found without folding.



$$A = \begin{array}{c} \begin{array}{ccccc} a & b & c & d & e \end{array} \\ \left[\begin{array}{ccccc} 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 \end{array}\right] \end{array} \begin{array}{c} \\ r1 \\ r2 \\ r3 \\ q1 \\ q2 \end{array}$$

(a) (b)

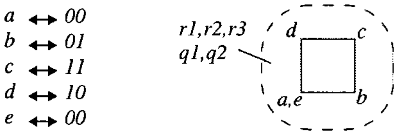
Fig. 13. (a) Relation matrix. (b) Corresponding affinity graph before folding.



$$A = \begin{array}{c} \begin{array}{cccc} {}^{a}_{e} & b & c & d \end{array} \\ \left[\begin{array}{cccc} 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{array}\right] \end{array} \begin{array}{c} \\ r1 \\ r2 \\ r3 \\ q1 \\ q2 \end{array}$$

(a) (b)

Fig. 14. (a) Relation matrix. (b) Corresponding affinity graph after folding a and e.



(a) (b)

Fig. 15. (a) Encoding after folding a and e. (b) Corresponding representation in the Boolean hyperspace.

The symbol a is in the points-to set of r1 and q1, whereas the symbol e is in the points-to set of r3 and q2. According to the pointer-dependence graph, these points-to sets are not dependent. The symbols associated with a and e can be folded. After folding, we end up with the graph in Fig. 14. This leads to an encoding that requires only 2 b (see Fig.15).

E. Encoding with Splitting

In the local encoding problem, one symbol can also have different codes in the different points-to sets.

Definition 10: We define splitting the action of assigning two or more codes to one symbol (or location set).

In Sections VI-C and VI-D, each location set was associated with a unique symbol that was encoded. After splitting, one location set may be associated with more than one symbol: splitting a symbol s_i is equivalent to creating a new symbol s_i', which corresponds to the same location set. The original symbol s_i and the newly created s_i' are then encoded into e_i and e_i', respectively.

Proposition 2: A points-to set Π_k that contains a symbol s_i may, after splitting s_i, contain the newly created symbol s_i' if and only if there is no code equal to e_i' in the encoded set E_k or in any encoded set dependent of Π_k.

Example 22: Let us consider the pointer-dependence graph in Fig. 16, where r1, r2, and r3 may, respectively, point to {a, b}, {b, c}, and {a, c}. The relation matrix and the corresponding affinity graph are presented in Fig. 17.

We would like to encode r1, r2, and r3 with 1 b and q with 2 b. We also want the codes of r1, r2, and r3 to be subfields of the code of q.

[5] A pair of vertices a, b in a graph are said to be *fused* (merged or identified) if the two vertices are replaced by a single vertex such that every edge that was incident on either a or b or on both is incident on the new vertex [13].

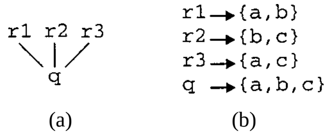
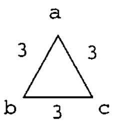


Fig. 16. (a) Pointer-dependence graph. (b) Definitions of the points-to sets.

$$A = \begin{array}{cccc} a & b & c & \\ 1 & 1 & 0 & r1 \\ 0 & 1 & 1 & r2 \\ 1 & 0 & 1 & r3 \\ 1 & 1 & 1 & q \end{array}$$

(a)

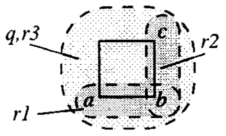


(b)

Fig. 17. (a) Relation matrix. (b) Corresponding affinity graph before splitting.

$a \leftrightarrow 00$
$b \leftrightarrow 01$
$c \leftrightarrow 11$

(a)

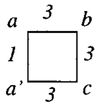


(b)

Fig. 18. (a) Encoding without splitting. (b) Corresponding representation in the Boolean hyperspace.

$$A = \begin{array}{ccccc} a & a' & b & c & \\ 1 & 0 & 1 & 0 & r1 \\ 0 & 0 & 1 & 1 & r2 \\ 0 & 1 & 0 & 1 & r3 \\ 1 & 1 & 1 & 1 & q \end{array}$$

(a)



(b)

Fig. 19. (a) Relation matrix. (b) Corresponding affinity graph after splitting symbol a.

Using the encoding technique without splitting symbols, we can find the encoding in Fig. 18. In this case, r1 and r2 are encoded on 1 b but the encoding of r3 requires 2 b.

After splitting the symbol a, we end up with the two symbols a and a′. The new encoding problem is presented in Fig. 19. We can find the encoding in Fig. 20, where the symbol a is in the points-to set of r1, r2, and q, and a′ in the points-to set of r3 and q.

The encoding in Fig. 20 is optimal: r1, r2, and r3 are encoded on 1 b and the assignments to q (q = r1, q = r2, q = r3) do not require any additional logic.

As described in Section VI-B, the symbols in each set can have different codes. Therefore, to minimize the dimension of the supercube of the encoded symbols in a points-to set [i.e., (1)], we can create new symbols associated with the same location sets for this points-to set. Note that if we split the symbols for each points-to set, we end up with a local encoding scheme close to the one presented in Section VI-B. The only difference is that one symbol may have multiple encodings within the same points-to set. However, to limit the increase in complexity, we are trying to split as few symbols as possible and only when useful to reduce the cost function.

When a symbol s_i is split, a new symbol s'_i is created. For each points-to set Π_k such that $s_i \in \Pi_k$, we decide whether the new points-to set Π'_k contains s_i, s'_i or both s_i and s'_i. The new set of encoded symbols E'_k can be defined as

$$E'_k = ((E_k - \{e_i\}) \cup E')$$

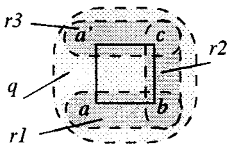


Fig. 20. Result of the encoding after splitting symbol a.

where

$$E' \text{ is either } \{e_i\}, \{e'_i\} \text{ or } \{e_i, e'_i\}. \tag{8}$$

In order to minimize (1), for every set Π'_k that may contain s_i or s'_i, we want to minimize

$$\dim(\text{supercube}(E'_k)) \tag{9}$$

which corresponds to

$$\min_{E'}(\dim(\text{supercube}((E_k = \{e_i\}) \cup E')))$$

where

$$E' \text{ is either } \{e_i\}, \{e'_i\} \text{ or } \{e_i, e'_i\}. \tag{10}$$

In the relation matrix $\boldsymbol{A}$, splitting is done by adding a column i' relative to s'_i. For each row α_k corresponding to a points-to set Π_k such that $s_i \in \Pi_k$, the pair of entries $(a^i_k, a^{i'}_k)$ is set to (0, 1), (1, 0), or (1, 1) according to (10). If (10) achieves its minimum for the three values $\{e_i, e'_i\}$, $\{e_i\}$, and $\{e'_i\}$, then we select $\{e_i, e'_i\}$. Example 23 illustrates the reason for this choice.

The new affinity graph can then be recomputed from the relation matrix. Splitting as well as folding can be incorporated in our graph-embedding algorithm as presented in Section VI-F.

Example 23: In Example 22 for the points-to set of r3, (10) is minimum for E' ={a'}; the dimension of the supercube of the encoded symbols in the new points-to set is minimum equal to one when it contains a′ only. As a result, in the relation matrix in Fig. 19, the entry a^1_3 is set to zero and a^2_3 is set to one. For the points-to set of q, (10) is minimum (equal to two) when E' is either {a}, {a′}, or {a, a′}. E'{a, a′} is then selected and the new points-to set of q contains both a and a′. Consequently, the entries a^1_4 and a^2_4 are both set to one. Since a is in the new points-to set of r1 and a′ in the new point-to set of r3, this allows us to implement both q = r3 and q = r1 trivially.

F. Encoding Algorithm

We propose a column-based approach such that the encoding matrix can be found column by column [10], [11], [14]. Our algorithm without folding and splitting is similar to the one used in Pow3 [3]. The pseudocode of the algorithm with folding and splitting is presented in Fig. 21.

The algorithm encodes the pointers with n b, where $n \geq \lceil \log_2(N) \rceil$. We consider one bit of the code at a time. For a symbol s_i associated with the code e_i, we consider the bits e^k_i for $k = \{1, 2, \ldots, n\}$. At each iteration k, we construct the kth column of the encoding matrix $\boldsymbol{E}$ by assigning bit e^k_i to all symbols for $i = \{1, 2, \ldots, N\}$. We ultimately want to distinguish all symbols. Therefore, in our algorithm, we have to make sure that at each iteration k, we have less than 2^{n-k} symbols associated with the same code. For example, for $k = (n - 1)$, we cannot have more than two symbols with the same code.

Definition 11: There is a class violation at iteration k when more than 2^{n-k} symbols have the same code so far.

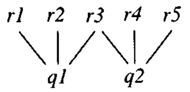
```
encode pointer(n) {
  /* construct matrix E one column at a time */
  for $k$=1 to n
    assign_code($k$);
}

assign_code(k) {
  sort edges by weight in decreasing order;
  foreach edge {$s_i$,$s_j$} {
    if($e_i^k$ and $e_j^k$ not assigned) {
      $e_i^k$ = $e_j^k$ = select_bit($s_i$,$s_j$);
      if(class violation) {
        ok=try_fold($s_i$);
        if(!ok) try_fold($s_j$); }
    }
    else if($s_i$ or $s_j$ not assigned) {
      $s_h$=unassigned($s_i$,$s_j$);$s_l$=assigned($s_i$,$s_j$);
      $e_h^k$ = $e_l^k$;
      if(class violation)
        try_fold($s_h$);
    }
    if($e_h^k$!=$e_l^k$)
    /* $s_i$ and $s_j$ already assigned or folding failed*/
      violated_edges->add({$s_i$,$s_j$})
  }
  sort violated edges by weight in decreasing order;
  foreach violated edge {$s_i$,$s_j$} {
    $s_h$=symbol whose sum of the weights on incident
edges is higher
    $s_l$=the other
      ok=try_split($s_h$);
      if(!ok) try_split($s_l$);
  }
}

bool try_split($s_i$) {
    create $s_i'$
    $e_i'$=$e_i$xor(1<<k);
    if(class violation)
      return try_fold($s_i'$);
    return false;
  }

bool try_fold($s_i$) {
    if($\exists s_j$s.t. Proposition 1 verified and $e_i$==$e_j$){
      fold($s_i$,$s_j$);
      remove $s_i$;
      return true;
    }
    return false;
  }
```

Fig. 21. Graph embedding algorithm with splitting and folding.

Note that at iteration k, we are only considering the k first bits of the codes since the other ones have not been assigned yet.

At each iteration k, e_i^k is defined for every symbol s_i. The assignment is done by considering the symbols on every edge endpoints starting with the edges with highest weights. The weights at each iteration are adjusted using the following formula [3]:

$$w_{i,j}^{\text{new}} = w_{i,j} \cdot (H(e_i, e_j) + 1) \tag{11}$$



(a)

$r1 \rightarrow \{a,b\}$
$r2 \rightarrow \{a,c\}$
$r3 \rightarrow \{b,c\}$
$r4 \rightarrow \{b,d\}$
$r5 \rightarrow \{c,d\}$

(b)

Fig. 22. (a) Pointer-dependence graph. (b) Definitions of the points-to sets.

$$A = \begin{array}{c c} \begin{array}{cccc} a & b & c & d \end{array} & \\ \left[\begin{array}{cccc} 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \end{array}\right] & \begin{array}{c} r1 \\ r2 \\ r3 \\ r4 \\ r5 \\ q1 \\ q2 \end{array} \end{array}$$

(a)

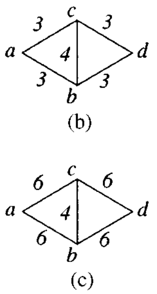


Fig. 23. (a) Relation matrix. (b) Affinity graph at the beginning of iteration 1. (c) Affinity graph at the beginning of iteration 2.

where $H(e_i, e_j)$ is the Hamming distance between the partially assigned codes of symbols s_i and s_j.

For the symbols incident to the edges $\{s_i, s_j\}$, we try to assign the same value to both e_i^k and e_j^k. However, this may not be possible in two cases. First, at each iteration of k, the number of symbols having the same code is limited to prevent class violations (cf. Definition 11). Moreover, if the symbols s_i and s_j are also incident to other edges whose weights are higher than $w_{i,j}$, they may already have been assigned two different values e_i^k and e_j^k. These two conditions are expressed below in Proposition 3.

Definition 12: An edge $\{s_i, s_j\}$ is said to be violated at iteration k if the bits e_i^k and e_j^k associated with the two symbols incident to the edge have different values.

Proposition 3: An $\{s_i, s_j\}$ is violated at iteration k if either one of the following conditions apply.

- There is class violation (and therefore, e_i^k and e_j^k need to have different values).
- Different values e_i^k and e_j^k have already been assigned to the two symbols.

In the case of a class violation, we try to fold one of the symbols on the edge $\{s_i, s_j\}$ with any of the previously assigned symbols. At this stage, two symbols are folded if Proposition 1 holds and if they have the same partial code so far.

If the edge $\{s_i, s_j\}$ is still violated (i.e., $e_i^k \neq e_j^k$), we try to split the symbols incident to the edge. One symbol can be split if the newly created symbol does not cause any class violation or can be folded with another symbol. In our algorithm, for a symbol s_i, we create a new symbol s_i' associated with a code e_i' such that $e'^l_i = e_i^l$ for $l < k$ and $e'^k_i = e_j^k \oplus 1$. In case of a class violation, we try to fold this new symbol. If folding cannot be done, the symbol s_i is not split.

Example 24: Consider the problem presented in Fig. 22. The associated relation matrix and affinity graph are presented in Fig. 23 in which pointer `q1` may take the value of `r1`, `r2`, or `r3` and `q2` may take the value of `r3`, `r4`, or `r5`.

Since we have four symbols, we want to encode them on $n = 2$ b. The encoding is computed in two iterations. After the first iteration, at most, two symbols can have the same encoding to prevent class violations.

At iteration $k = 1$, we first take the edge with the highest weight {`b, c`} and assign the value 0 to `b` and `c`. Since we want the code to be 2-b long, we can have at most two symbols with the same code after the first iteration. The value `1` is, therefore, assigned to `a` and `d` and all edges beside {`b, c`} are violated. We then try to fold the symbols. Folding cannot be performed. For example, for the edge {`a, b`}, `a` cannot be folded with `b` because both symbols are in the points-to set of `r1` and `q1`. Symbol `a` cannot be folded with `c` either because both symbols are in the points-to set of `r2` and `q1`. The violated edges are {`a, b`}, {`a, c`}, {`d, c`}, and {`d, b`}. We then try to split the symbols on these edges. Splitting cannot be performed either. For example, when we try to split variable `a`, we create a new variable `a′` with code 0 and the following relation matrix is computed:

$$A = \begin{array}{cccccl} a & a' & b & c & d & \\ 0 & 1 & 1 & 0 & 0 & r1 \\ 0 & 1 & 0 & 1 & 0 & r2 \\ 0 & 0 & 1 & 1 & 0 & r3 \\ 0 & 0 & 1 & 0 & 1 & r4 \\ 0 & 0 & 0 & 1 & 1 & r5 \\ 0 & 1 & 1 & 1 & 0 & q1 \\ 0 & 0 & 1 & 1 & 1 & q2 \end{array} . \tag{12}$$

We have three variables {`a′, b, c`} with the same code 0, which creates a class violation. We then try to fold `a′` with `b` or `c`. This cannot be done because `a′` and `b` are in the points-to sets of `r1` and `q1` and `a′` and `c` are both in the points-to sets of `r2` and `q1`. As a result, the encoding after the first iteration is `0` for `b` and `c` and `1` for `a` and `d`. At iteration $k = 2$, we assign the value 0 to `b`, 1 to `c`, 0 to `a`, and 1 to `d`. Note that other values could be assigned depending on the order in which edges of equal weight are taken in the implementation. All edges are violated. Among the edges with maximum weight are {`a, c`} and {`b, d`}. We try to split `a` on the edge {`a, c`} and create the new symbol `a′`. The resulting relation matrix is

$$A = \begin{array}{cccccl} a & a' & b & c & d & \\ 1 & 0 & 1 & 0 & 0 & r1 \\ 0 & 1 & 0 & 1 & 0 & r2 \\ 0 & 0 & 1 & 1 & 0 & r3 \\ 0 & 0 & 1 & 0 & 1 & r4 \\ 0 & 0 & 0 & 1 & 1 & r5 \\ 1 & 1 & 1 & 1 & 0 & q1 \\ 0 & 0 & 1 & 1 & 1 & q2 \end{array} . \tag{13}$$

Variable `a′` can be folded with `d` because Proposition 1 holds: `a′` and `d` have the same code at the previous iteration and are not elements of dependent points-to sets. After folding, we end up with the following relation matrix:

$$A = \begin{array}{ccccl} & a' & & & \\ a & d & b & c & \\ 1 & 0 & 1 & 0 & r1 \\ 0 & 1 & 0 & 1 & r2 \\ 0 & 0 & 1 & 1 & r3 \\ 0 & 1 & 1 & 0 & r4 \\ 0 & 1 & 0 & 1 & r5 \\ 1 & 1 & 1 & 1 & q1 \\ 0 & 1 & 1 & 1 & q2 \end{array} . \tag{14}$$

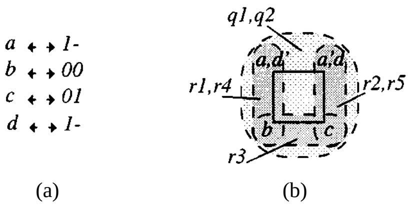


Fig. 24. (a) Encoding after splitting and folding (where "–" is a *do not care*. b) Corresponding representation in the Boolean hyperspace.

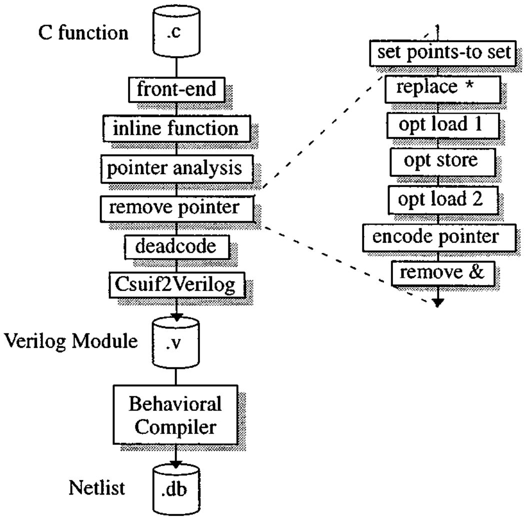


Fig. 25. Toolflow for the synthesis of pointer in C.

The variable `d` (which is now mapped to a symbol representing both `d` and `a′`) can also be split and the new symbol `d′` can be folded with `a`. The final relation matrix is then

$$A = \begin{array}{ccccl} d' & a' & & & \\ a & d & b & c & \\ 1 & 0 & 1 & 0 & r1 \\ 0 & 1 & 0 & 1 & r2 \\ 0 & 0 & 1 & 1 & r3 \\ 1 & 0 & 1 & 0 & r4 \\ 0 & 1 & 0 & 1 & r5 \\ 1 & 1 & 1 & 1 & q1 \\ 1 & 1 & 1 & 1 & q2 \end{array} . \tag{15}$$

We end up with the encoding in Fig. 24 in which all constraints are satisfied.

VII. Implementation

We have implemented the different algorithms using the SUIF environment [52], [66]. The toolflow is presented in Fig. 25. Our implementation takes a function with pointers in C and generates a module in Verilog. This module can then be synthesized using the behavioral compiler [67]. For hardware synthesis, the timing information is expressed in the C model: `wait()` in C will be translated into @(`posedge clk`) in Verilog. The ports and the data types are defined in a separate header file. The translation from C to Verilog consists of different passes. After the front end, we inline the functions and perform the pointer analysis [50]. Then the points-to information is used to remove and optimize pointers in the following order:

— define the points-to set of each pointer;

— replace the loads and stores (insert `star_p` and `tmp_p`);

— optimize load 1: define `star_p` when `p` or any variable of the points-to set change;
— optimize loads followed by stores: create the `_starN_p` variables;
— optimize load 2: kill `star_p` when all variables of the points-to set are live;
— encode pointers' value;
— dead-code elimination.

The intermediate code without pointers is then translated into Verilog using Csuif2Verilog.

We have recently ported our research to the Synopsys Cocentric SystemC compiler [68] to synthesize C models into hardware directly, without having to translate C into HDL. In addition, we have also developed a tool to implement dynamic memory allocation in hardware [44].

VIII. Results

We first show the results for the resolution of pointers in relatively large examples. Then we illustrate the effect of pointers' encoding and of the optimization of loads and stores on selected examples.

Since there are no synthesis benchmarks written in C with pointers, the objective of this section is to show the technical feasibility of mapping C descriptions to logic gates. In order to test our tool on real examples, we present the implementation of two algorithm: a two-dimensional inverse discrete cosine transform (2-D IDCT) [31] and an alpha blender written in C. The 2-D IDCT is widely used in image compression standards such as JPEG, MPEG, and H263. The 2-D IDCT implemented consists of two one-dimensional inverse discrete cosine transforms (1-D IDCTs). For this purpose, we use three different memories: the input buffer (`in_table`), the intermediate buffer that stores the result of the first 1-D IDCT (`buf_table`), and the output buffer (`out_table`). These memories are accessed through pointers and pointer arithmetic. Pointers are also used in the 1-D IDCT to reference two register banks (`buff1` and `buff2`).

The 2-D IDCT is implemented using only one call to 1-D IDCT (function `1d_idct`), which is inlined before synthesis

```
2d_idct () {
  int i, ⋆ p_ in, ⋆ p_out;
  for(i = 0; i < 2; i++) {
    if(i == 0) {
      // first iteration
        p_in = in_table;
      // p_in → input buffer
        p_out = buf_table;
      // p_out → intermediate buffer
    } else {
      // second iteration
        p_in = buf_table;
      // p_in → intermediate buffer
        p_out = out_table;
      // p_out → output_buffer
    }
    1d_idct(p_in, p_out);
      // unique call to 1D IDCT
  }
}.
```

TABLE I
Result of the Synthesis of the IDCT Running at 20 MHz Using Target Library lsi_10k (Area in Library Units)

test	cpu time	nb ptr	C lines	Verilog lines	area		cycles at 20MHz
					combinational	non-combin.	
idct	7.8s	6	176	221	38,172	12,910	6
alpha blender	10.2s	9	119	189	123,750	149,350	8

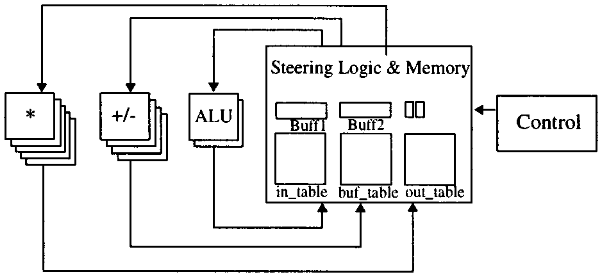


Fig. 26. Architecture of the 2-D IDCT.

Note that in this specific example, pointers are not only used to access memories, but they are also used for sharing resources. In this example, only one `1d_idct` is synthesized. Since functions are inlined in our framework, a more standard implementation of the 2-D IDCT algorithm, in which the `1d_idct` function is called twice, would lead to two 1-D IDCT blocks. Such a design would typically be larger and more difficult to efficiently synthesize. Using pointers here provides a convenient and efficient way of performing resource sharing.

The second example corresponds to an alpha blender. Alpha blenders are used in video and signal processing to superimpose multiple images. Our implementation takes three images and alpha planes of size 8 × 8. The alpha plane defines the degree of opacity for each pixel in the image. The order in which the images are placed with respect to each other (e.g., front, middle, back) is defined by a layer number associated with each image. The different images and alpha planes are stored in separate arrays (mapped to separate memories) in order to access them in parallel. Pointers are used to access the different arrays.

The results after synthesis are presented on Table I. The central processing unit (CPU) time for translating the C model into Verilog was calculated on SunUltra2. The Verilog modules were synthesized with Behavior Compiler without unrolling loops. The architecture of the IDCT is presented in Fig. 26. The design consists of five multipliers, four adders, and two arithemetic and logic units (ALUs). Other implementations can be found by changing the timing and resource constraints.

We have written several models to study the effects of the different optimizations presented in Sections V and VI. These optimizations consist of encoding the pointers' value and reducing the number of live variables before loads and between loads and stores.

The first set of results illustrates the effects of each feature of the optimizer. Tables II and III show the examples with the area and cumulative timing after pointer resolution with and without optimization.

TABLE II
AREA AFTER SYNTHESIS AND OPTIMIZATION USING TARGET LIBRARY lsi_10k (AREA IN LIBRARY UNITS)

example	C lines	area (combinational/non-combinational)	
		no optimization	with optimizations
load	43	3861 (1527/2334)	3599 (2076/1523)
load/store	48	6746 (5319/1427)	6366 (5324/1042)
encoding	58	1106 (272/834)	996 (162/834)

TABLE III
TIMING AFTER SYNTHESIS AND OPTIMIZATION USING TARGET LIBRARY lsi_10k (IN ns)

example	C lines	timing	
		no optimization	with optimizations
load	43	46 ns	51 ns
load/store	48	86 ns	88 ns
encoding	58	7.5 ns	5.9 ns

TABLE IV
AREA AFTER SYNTHESIS AND OPTIMIZATION USING tsmc.35 LIBRARY (IN LIBRARY UNITS). FOR EACH EXAMPLE, P REPRESENTS THE NUMBER OF POINTERS AND N THE NUMBER OF VARIABLES

example (P/N)	encoding	storage	assignment	load/store	total
test1 (5/5)	global	5,512	1,231	11,631	18,374
	simple-alg	3,307	793	9,768	13,868
	split&fold	2,756	712	8,456	11,924
	min-length	2,756	1,134	8,391	12,281
	1-hot	4,685	1,474	7,354	13,513
test2 (7/4)	global	3,582	1,020	14,256	18,858
	simple-alg	3,047	988	14,591	18,626
	split&fold	2,480	842	12,976	16,298
	min-length	2,480	1,020	13,041	16,541
	1-hot	4,409	1,490	12,668	18,567
test3 (9/7)	global	7,716	2,705	30,731	41,152
	simple-alg	5,236	2,203	28,479	35,918
	split&fold	4,961	2,122	28,220	35,303
	min-length	4,961	3,240	28,042	36,243
	1-hot	8,543	5,686	25,579	39,808

The first model (`load`) tests the optimization of loads. It contains one pointer that may point to three integers stored in registers. After the definition of the pointer, we have two paths and then a load. In one path, none of the variables of the points-to set are used. In the other path, all variables of the points-to set become live. Without any optimization, we have five 32 b registers (i.e., 2334 units of noncombinational area). After optimization, the number of registers is reduced to three (i.e., 1523 units of noncombinational area). This reduction of the storage goes with an increase of the combinational area and of the cumulative timing caused by adding steering logic to update the value of `star_p`. There is a tradeoff between the number of registers and the size of the steering logic.

In the second example (`load/store`), we have a pointer that may point to two integer variables stored in registers. This pointer is used as a parameter in a function call. After inlining the function, we end up with a load followed by a store. Here the optimization saves one register with a little increase of the amount of steering logic.

Finally, the last example (`encoding`) implements the model described in Example 15 with the two encodings presented in Example 16. Here the encoding of the pointers value reduces the combinational logic by 40%. Since the design is simpler, the circuit is also faster.

The second set of examples compares our encoding algorithm to other encoding schemes. The results are presented in Table IV. They have been obtained as follows. Pointers' encoding has effect on three components of the design: the number of registers necessary to store the pointers' value (`storage`), the logic necessary to assign and compare pointers (`assignment`), and the implementation of loads and stores (`load/store`). Each of these components is synthesized using Synopsys design compiler. We present the results for five different schemes.

First, we present the results for a global encoding (`global`) in which we associate the same code with all symbols associated to the same variable in the different points-to sets. In this case, assignments or comparisons of pointers can be performed without translating the values of the pointers. However, the number of bits used for the encoding is not minimal, which leads to larger decoding circuits (cf. both `load/store` and `assignment`) and more registers (cf. `storage`).

The second scheme (`simple-alg`) is the implementation of the heuristic algorithm presented in Section VI without splitting and folding. The size of the pointer is then reduced but is still not always minimal. The results for the algorithm with folding and splitting (`split&fold`) are given. The length of the codes is then close to the minimum and the size of the combinational circuit for both `assignment` and `load/store` is reduced, which gives better results.

Results for minimum-length encoding (`min-length`) are also given. In this suboptimal encoding (similar to the nonoptimal encoding used in Example 16), each variable in each points-to set is simply associated with a number (zero for the first variable, one for the second variable, etc. ...). The number of bits used to encode each tag is then minimum but the size of the circuit that translates the values of the pointers is not. Finally, one-hot (`1-hot`) encoding gives larger codes. However, the specific proprieties of the resulting codes can be used to simplify the decoding logic, especially in loads and stores.

In this section, we have shown how C code with pointer variables can be synthesized by removing the pointers and using high-level synthesis. Moreover, variations on the implementation may be explored using the optimizations presented in Sections V and VI. Even though the effect of these optimizations may be limited in general, they can be used to reduce the storage areas and/or the steering logic. In particular, optimization of loads and stores can be used to reduce the number of registers with an increase on the amount of steering logic. Encoding, on the other hand, can be used to reduce both the size of the pointers and the logic necessary to translate and decode the pointers' value, leading to better performances.

IX. Conclusion

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

Our toolflow fits into current methodology and supports the mapping of data to multiple memories, registers, or wires. Compiler techniques are used to reduce the storage before pointer loads and stores. Heuristics are used to efficiently encode the values of pointers by reducing their size and by optimizing the circuits implementing assignments and comparisons of pointers.

The synthesis of pointers raises the level of abstraction at the input of high-level synthesis. Models can be described at the behavioral level using the notions of a single address space and of indirect memory references found in many programming languages. The techniques and optimizations presented here can be generalized to support more of the $C/C{++}$ syntax as well as other programming languages, facilitating the mapping of functions and complex data structures including object-oriented features into hardware.

Acknowledgment

The authors would like to thank D. Heine of Stanford University, Stanford, CA, for his help with the implementation using SUIF and J. Kunkel and A. Ghosh of Synopsys Inc., Mountain View, CA, for their comments and support. The authors would also like to thank the anonymous reviewers for their comments and suggestions.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers, Principles, Techniques, and Tools*. Reading, MA: Addison-Wesley, 1986.

[2] J. Babb, M. Rinard, A. Moritz, W. Lee, M. Frank, R. Barua, and S. Amarasinghe, "Parallelizing applications into silicon," in *Proc. IEEE Workshop FPGAs Custom Computing Machines*, Napa, CA, Apr. 1999.

[3] L. Benini and G. De Micheli, "State assignment for low power dissipation," *IEEE J. Solid-State Circuits*, vol. 11, pp. 258–268, Mar. 1995.

[4] L. Benini, G. De Micheli, E. Macii, D. Sciuto, and C. Silvano, "Address bus encoding techniques for system-level power optimization," in *Proc. Design Automation Test Eur.*, Paris, France, Feb. 1998, pp. 861–866.

[5] I. Bolsens, H. J. De Man, B. Lin, K. Van Rompaey, S. Vercauteren, and D. Verkest, "Hardware/software codesign of digital telecommunication systems," *Proc. IEEE*, vol. 85, pp. 391–418, Mar. 1997.

[6] C. T. Bye, M. R. Lightner, and D. L. Ravenscroft, "A functional modeling and simulation environment based on ESIM and C," in *Proc. Int. Conf. Computer-Aided Design*, Nov. 1984, pp. 51–53.

[7] T. J. Callahan and J. Wawrzynek, "Instruction-level parallelism for reconfigurable computing," in *Proc. 8th Int. Workshop Field Programmable Logic Applications*, Berlin, Germany, 1998, pp. 248–257.

[8] R. Composano and W. Wolf, *High-Level VLSI Synthesis*. Norwell, MA: Kluwer, 1991.

[9] F. Catthoor, S. Wuytack, E. De Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle, *Custom Memory Management Methodology*. Norwell, MA: Kluwer, 1998.

[10] G. De Micheli, *Synthesis and Optimization of Digital Circuits*, New York: McGraw-Hill, 1994.

[11] ——, "Symbolic design of combinational and sequential logic circuits implemented by two-level logic macros," *IEEE Trans. Computer-Aided Design*, vol. CAD-5, pp. 597–616, 1986.

[12] ——, "Hardware synthesis from $C/C{++}$," in *Proc. Design Automation Test Eur.*, Munich, Germany, pp. 382–383.

[13] N. Deo, *Graph Theory with Applications to Engineering and Computer Science*. Englewood Cliffs, NJ: Prentice-Hall, 1974.

[14] T. A. Dolotta and E. J. McCluskey, "The encoding of internal states of sequential machines," *IEEE Trans. Electron. Comput.*, vol. EC-13, pp. 549–562, Oct. 1964.

[15] C. Duff, "Codage d'automates et theorie des cubes intersectants," Ph.D. dissertation, Inst. Nat. Polytech. de Grenoble, Grenoble, France, Mar. 1991.

[16] R. Ernst, J. Henkel, T. Benner, W. Ye, U. Holtmann, and M. Trawny, "The COSYMA environment for hardware/software cosynthesis of small embedded systems," *Microprocess. Microsyst.*, vol. 20, no. 3, pp. 159–166, May 1996.

[17] D. Gajski, F. Vahid, S. Narayan, and J. Gong, *Specification and Design of Embedded Systems*. Englewood Cliffs, NJ: Prentice-Hall, 1994.

[18] D. Gajski, N. Dutt, A. Wu, and S. Lin, *High-Level Synthesis, Introduction to Chip and System Design*. Norwell, MA: Kluwer, 1992.

[19] A. Ghosh, J. Kunkel, and S. Liao, "Hardware synthesis from $C/C{++}$," in *Proc. Design Automation Test Eur.*, Munich, Germany, pp. 387–389.

[20] M. B. Gokhale and R. Minnich, "FPGA computing in a data parallel C," in *Proc. IEEE Workshop FPGAs Custom Computing Machines*, Napa, CA, 1993, pp. 94–101.

[21] E. Goldberg, T. Villa, R. Brayton, and A. Sangiovanni-Vincentelli, "Theory and algorithms for face hypercube embedding," *IEEE Trans. Computer-Aided Design*, vol. 17, pp. 472–488, June 1998.

[22] 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*, Orlando, FL, May 1999.

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

[24] H. Kim and K. Choi, "Transformation from C to synthesizable VHDL," in *Proc. Asia Pacific Conf. HDL APCHDL*, July 1998, pp. 85–88.

[25] D. Knapp, *Behavioral Synthesis: Digital System Design Using the Synopsys Design Compiler*. Englewood Cliffs, NJ: Prentice-Hall, 1996.

[26] D. Ku and G. De Micheli, *High-Level Synthesis of ASICs Under Timing and Synchronization Constraints*. Norwell, MA: Kluwer, 1992.

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

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

[29] S. Liao, "Toward a new standard for system level design," in *Proc. 8th Int. Workshop Hardware–Software Codesign*, San Diego, CA, May 2000, pp. 2–6.

[30] S.-W. Liao, A. Diwan, R. P. Bosch Jr., A. Ghuloum, and M. S. Lam, "SUIF explorer: An interactive and interprocedural parallelizer," in *Proc. 7th ACM SIGPLAN Symp. Principles Practice Parallel Program.*, May 1999, pp. 37–48.

[31] E. Linzer and E. Reig, "New scaled DCT algorithms for fused multiply/add architectures," in *Proc. Int. Conf. Acoustics, Speech, Signal Processing*, vol. 1–5, 1991, pp. 2201–2204.

[32] Y. Li, T. Callahan, E. Darnell, R. Harr, U. Kurkure, and J. Stockwood, "Hardware-software codesign of embedded reconfigurable architectures," in *Proc. Design Automation Conf.*, Los Angeles, CA, June 2000, pp. 507–512.

[33] O. Mencer, M. Morf, and J. Flynn. PAM-Blox, high-performance FPGA design for adaptive computing. presented at IEEE Symp. FPGAs Custom Computing Machines. [Online]. Available: http://umunhum.stanford.edu/PAM-Blox

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

[35] A. R. Newton, S. Devaras, H.-K. Ma, and A. Sangiovanni-Vincentelli, "MUSTANG: State assignment of finite state machines targeting multilevel logic implementations," *IEEE Trans. Computer-Aided Design*, vol. 7, pp. 1290–1300, Dec. 1988.

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

[37] R. Rugina and M. Rinard, "Pointer analysis for multithreaded programs," in *Proc. SIGPLAN Conf. Program Language Design Implementation*, Atlanta, GA, May 1999, pp. 77–90.

[38] G. Saucier, "State assignment of asynchronous sequential machines using graph techniques," *IEEE Trans. Computer*, Mar. 1972.

[39] G. Saucier, C. Duff, and F. Poirot, "State assignment using a new embedding method based on intersecting cube theory," in *Proc. Design Automation Conf.*, Las Vegas, NV, June 1989, pp. 321–326.

[40] G. Saucier, M. C. Depaulet, and P. Sicard, "ASYL: A rule-based system for controller synthesis," *IEEE Trans. Computer-Aided Design*, vol. CAD-6, pp. 1088–1097, Nov. 1987.

[41] 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.*, San Francisco, CA, June 1998, pp. 315–320.

[42] L. Séméria and G. De Micheli, "SpC: Synthesis of pointers in C, application of pointer analysis to the behavioral synthesis from C," in *Proc. IEEE/ACM Int. Conf. Computer-Aided Design*, San Jose, CA, Nov. 1998, pp. 340–346.

[43] ——, "Encoding of pointers for hardware synthesis," in *Proc. Int. Workshop IP-Based Synthesis Syst. Design*, Grenoble, France, Dec. 1998, pp. 57–63.

[44] 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 Test Eur.*, Paris, France, Mar. 2000, pp. 312–319.

[45] ——, "Memory representation and hardware synthesis of C code with pointers and complex data structures," in *Proc. Synthesis and Systems Integration Mixed Technologies Workshop*, Kyoto, Japan, Apr. 2000, pp. 43–48.

[46] B. Steensgaard, "Points-to analysis by type inference of programs with structures and unions," in *Proc. Int. Conf. Compiler Construction*, Berlin, Germany, Apr. 1996, pp. 136–150.

[47] C. Stoud, R. Munoz, and D. Pierce, "Behavioral model synthesis with cones," *IEEE Design Test Comput.*, vol. 5, pp. 22–30, June 1988.

[48] T. Villa and A. Sangiovanni-Vincentelli, "NOVA: State assignment of finite state machines for optimal two-level logic implementation," *IEEE Trans. Computer-Aided Design*, vol. 9, pp. 905–924, Sept. 1990.

[49] K. Wakabayashi, "C-based synthesis with behavioral synthesizer, cyber," in *Proc. Design Automation Test Eur.*, Munich, Germany, 1999, pp. 390–391.

[50] R. Wilson, "Efficient, context-sensitive pointer analysis For C programs," Ph.D. dissertation, Stanford University, Stanford, CA, 1997.

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

[52] R. P. Wilson , R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. Tjiang, S. W. Liao, C. W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy, "SUIF: An infrastructure for research on parallelizing and optimizing compilers," *ACM SIPLAN Notices*, vol. 28, no. 9, pp. 67–70, Sept. 1994.

[53] S. Wuytack, J. L. 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.

[54] C Level Design [Online]. Available: http://www.cleveldesign.com

[55] CoWare N2C [Online]. Available: http://www.coware.com/n2c.html

[56] CynApps [Online]. Available: http://www.cynapps.com

[57] EtherDesign Software [Online]. Available: http://www.etherdesign.com

[58] Frontier Design AR|T BUILDER [Online]. Available: http://www.frontierd.com/artbuilder.htm

[59] Handle-C [Online]. Available: http://oldwww.comlab.ox.ac.uk/oucl/groups/hwcweb/handel/index.html

[60] Mentor Graphics Monet [Online]. Available: http://www.mentor.com/monet

[61] Get2Chip [Online]. Available: http://www.get2chip.com

[62] OCAPI [Online]. Available: http://www.imec.be/ocapi

[63] LCLint [Online]. Available: http://lclint.cs.virginia.edu

[64] Rational Software Purify [Online]. Available: http://www.rational.com

[65] SpecC [Online]. Available: http://www.specc.gr.jp

[66] SUIF Compiler Framework [Online]. Available: http://suif.stanford.edu

[67] Synopsys Inc. Behavioral Compiler [Online]. Available: http://www.synopsys.com

[68] Synopsys Inc. CoCentric SystemC Compiler [Online]. Available: _http://www.synopsys.com/products/cocentric_systemC

[69] SystemC [Online]. Available: http://www.systemc.org

[70] Transmogrifier [Online]. Available: http://www.eecg.toronto.edu/EECG/RESEARCH/tmcc/tmcc/

[71] RAW Architecture [Online]. Available: http://www.cag.lcs.mit.edu/raw

Luc Séméria received the Engineer degree from the École Nationale Supérieure des Télécommunications, Paris, France, in 1996 and the M.S. degree in electrical engineering from Stanford University, Stanford, CA, in 1998. He is working toward the Ph.D. degree in the Electrical Engineering Department, Stanford University.

While studying, he has held several summer positions at Synopsys Inc., Mountain View, CA. His research interests include system-level design, hardware–software codesign, and optimizing compiler. He is currently working on the synthesis of hardware from C.

Giovanni De Micheli (S'79–M'83–SM'89–F'94) received the Nuclear Engineer degree from the Politecnico di Milano, Milan, Italy, in 1979 and the M.S. and Ph.D. degrees in electrical engineering and computer science from the University of California, Berkeley, in 1980 and 1983, respectively.

He is a Professor of Electrical Engineering and Computer Science at Stanford University, Stanford, CA. He was Codirector of the NATO Advanced Study Institutes on Hardware–Software Codesign, Tremezzo, Italy, in 1995 and the Logic Synthesis and Silicon Compilation, L'Aquila, Italy, in 1986. He is author of *Synthesis and Optimization of Digital Circuits* (New York: McGraw-Hill, 1994) and coauthor of four other books. 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.

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