APPLYING POINTER ANALYSIS TO THE SYNTHESIS OF HARDWARE FROM C

A DISSERTATION

SUBMITTED TO THE DEPARTMENT OF

ELECTRICAL ENGINEERING

OF STANFORD UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

Luc Séméria

June 2001

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

_____

Giovanni De Micheli, Principal Advisor

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

_____

Monica Lam

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

_____

Michael Flynn

Approved for the University Committee on Graduate Studies:

_____

ABSTRACT

With the advances in Computer Aided Design (CAD) technology, the design of digital circuits is becoming more and more as the development of software. Hardware is modeled using Hardware Description Languages (HDLs) very much as software is described using programming languages. Synthesis tools are used like compilers to map these HDL models into hardware. However, modern systems, which consist of mixed software/hardware modules, are often initially modeled using programming languages instead of HDLs. Specifically, C/C++-based languages with hardware support are used to quickly verify the functionality and estimate the performances at the system level.

One of the greatest challenges in C/C++-based design methodology is to efficiently map C/C++ models into hardware. Many of the networking and multimedia applications implemented in hardware or mixed hardware/software systems are making use of complex data structures stored in one or multiple memories. As a result, many of the C/C++ fea-

tures that were originally designed for software applications are now making their way into hardware. Such features include dynamic memory allocation and pointers used to manage data.

In this thesis, I present a solution for efficiently mapping arbitrary C code with pointers and `malloc`/`free` into hardware. 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. The values of the pointers are then encoded, and branching statements are used to dynamically access data referenced by pointers. Dynamic memory allocation and deallocation are supported by instantiating hardware memory allocators tailored to an application and a memory architecture. Several optimizations may also be performed. A heuristic algorithm is presented to efficiently encode the values of the pointers. Compiler techniques may also be used to reduce storage before loads and stores.

An implementation using the SUIF compiler framework is presented, followed by some examples of implementations taken from multimedia and networking applications.

ACKNOWLEDGMENTS

I would like thank my advisor Giovanni De Micheli for giving me the opportunity of working on this thesis, for his support and his directions. This work couldn't have been possible without the advises of several other people. First, I would like to acknowledge the support of Synopsys Inc. that initiated this research and financed it. Specifically, I would like to thank Raul Camposano, Abhijit Ghosh, Joachim Kunkel and Don Mc Millan for their support. I would also like to thank many of my friends met at Synopsys whose comments helped me at the different stages of this research: Olivier Coudert, Maurizio Damiani, Stan Liao, Preeti Panda, Srinivas Raghvendra, Mike Sharp. This research couldn't have been possible without the help of the SUIF group at Stanford and more specifically Monica Lam and Dave Heine. Finally, thank you to Koichi Sato from NEC corp., who was a visiting researcher at Stanford in 1999 and worked with me on the synthesis of dynamic memory allocation and deallocation.

A thesis is more than its technical contributions, it is also a personal achievement. For this I would warmly thank my family and friends. First my wife Cécile who joined me in this adventure. Thank you for your loving trust and your support. I would also like to thank my family here, Bernard and Joan, and in France, my parents, my brother and sisters whose encouragements throughout my studies have always been the greatest help.

This is also a good opportunity to thank some of the people who really made a difference in my student life. Jean-Marie Fluchaire, high-school mathematics professor at the Lycée du Grésivaudan Meylan, and Henri Koen, professor of mathematics in Classes Préparatoires at Lycée Masséna Nice.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF ILLUSTRATIONS

CHAPTER 1. INTRODUCTION

In the last decades, the complexity of digital systems implemented on silicon has grown at the rate of 2x every 16 months according to Moore's law. Within 5 years, integrated circuits (IC) will include as many as 190 million transistors [82]. However, the productivity of designers does not improve at the same rate. Studies have shown that designers or programmers write in average 10 lines of code per day. In order to bridge this productivity gap, Computer Aided Design (CAD) tools and methodologies have been developed to facilitate the task of designing systems on silicon. CAD tools enable more efficient system modeling by abstracting some of the implementation details that can be automatically derived from the specification.

The most dramatic achievements are for the design of digital circuits. Since the seventies, the placement and the connections of standard logic cells from a netlist onto silicon have been automated in *physical design* (i.e. Place & Route) *tools*. The eighties have seen the development of *logic synthesis* and the release of tools that automatically generate a netlist from a cycle-accurate behavioral model expressed using *Hardware Description Languages* (HDLs). Examples of HDLs include Verilog HDL [44] and VHDL [48]. In the

last decade *high-level synthesis* (HLS) *tools* (aka *behavioral* or *architectural synthesis tools*) have been developed to automate the mapping of sequential behavioral descriptions into cycle-accurate representations. Such tools perform both scheduling of operations (arithmetic and logic operations as well as a memory and register accesses) and resource binding [14].

1.1 Motivations

Different languages have been used as input to high-level synthesis. HDLs are the most commonly used. However, designers often write system-level models using programming languages, such as C or C++, to estimate the system performances and verify the functional correctness of the design. C and C++ offer many advantages. The first motivation for using programming languages, as opposed to HDLs at the system level, is the growing amount of software running on any system. Many of today's chips incorporate processor cores running instruction codes compiled from programming languages. Moreover, among programming languages, C and C++ have been the most widely used in the last two decades. As a result there is a vast amount of legacy code and libraries that can be reused to quickly model systems.

Even for hardware applications, C and C++ have been often used to accelerate the design process. C/C++ can be efficiently compiled onto today's architectures and thus are used to develop fast simulation models (e.g. to model microprocessors and microcontrolers). Besides, describing both the software and hardware in the same language facilitates the integration and the verification of hardware components within a software environ-

ment. Recent initiatives [71,79,86,89] attempt to standardize a C/C++-based language for both hardware and software design.

As a result, today, C/C++ or C/C++-based languages can be used to model systems for both software and hardware components. However, in order to physically map functionality onto hardware, one usually needed to manually translate C/C++ code into HDLs. This task is well-known for being both time consuming and error prone. The synthesis of hardware directly from C/C++ would then accelerate the design process. A coding style and a set of restrictions on the language are often defined to simplify the mapping of such models onto hardware.

Some of the components originally implemented in software may be also be mapped to specific hardware components for better performances (increased throughput or reduced latency), power savings, and/or smaller silicon area. To facilitate such software/hardware migration, synthesis tools would need to support all C/C++ language constructs.

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 [32,48]. C was originally designed to develop the UNIX operating system. It provides constructs to directly access memory (through pointers) and to manage memories and I/O using the standard C library (`malloc`, `free`,...). These constructs are widely used in software. Nevertheless, many of the networking and multimedia applications implemented in hardware or mixed hardware/software systems are also using complex data structures stored in one or multiple memory banks. As a result, many of the C/C++ features that were originally designed for software applications are now making their way into hardware.

The synthesis of such constructs as dynamic memory allocation, function calls, recursions, `goto`'s, type castings and pointers, turns out to be particularly challenging.

1.2 Design methodology

The general goal of this research is to automate the generation of digital circuits from an algorithmic or behavioral description written in C. The realization of this research is a prototyping tool which fits today's design methodology by taking a C function as an input and generating hardware automatically. To better understand where this research fits, it is important at this point to review the different steps involved in the design of a system. In the top-down approach shown on Figure 1.1, the system is first modeled at the system level using programming languages or more application-specific descriptions (such as Matlab, Mathematica, etc.). The models used at this level are designed to verify the functionality and estimate the performances of the system. Several optimizations are usually performed. They do not depend on whether the final implementation is hardware or software. Algorithmic optimizations are performed, for example to reduce the number of operations in a given computation. In addition, for a given algorithm, data formats are refined, for example from floating point to fixed point. Data structures are also refined. In particular, different implementations of queues or of hash tables could be evaluated.

At the architectural level, the system usually consists of a set of communicating processes or threads. These processes can either be mapped to hardware or software. The communication between the different processes is refined at this point. On the software side, drivers are generated. On the hardware side, controllers are synthesized to implement communication protocols. Internal and shared storages are defined for each process.

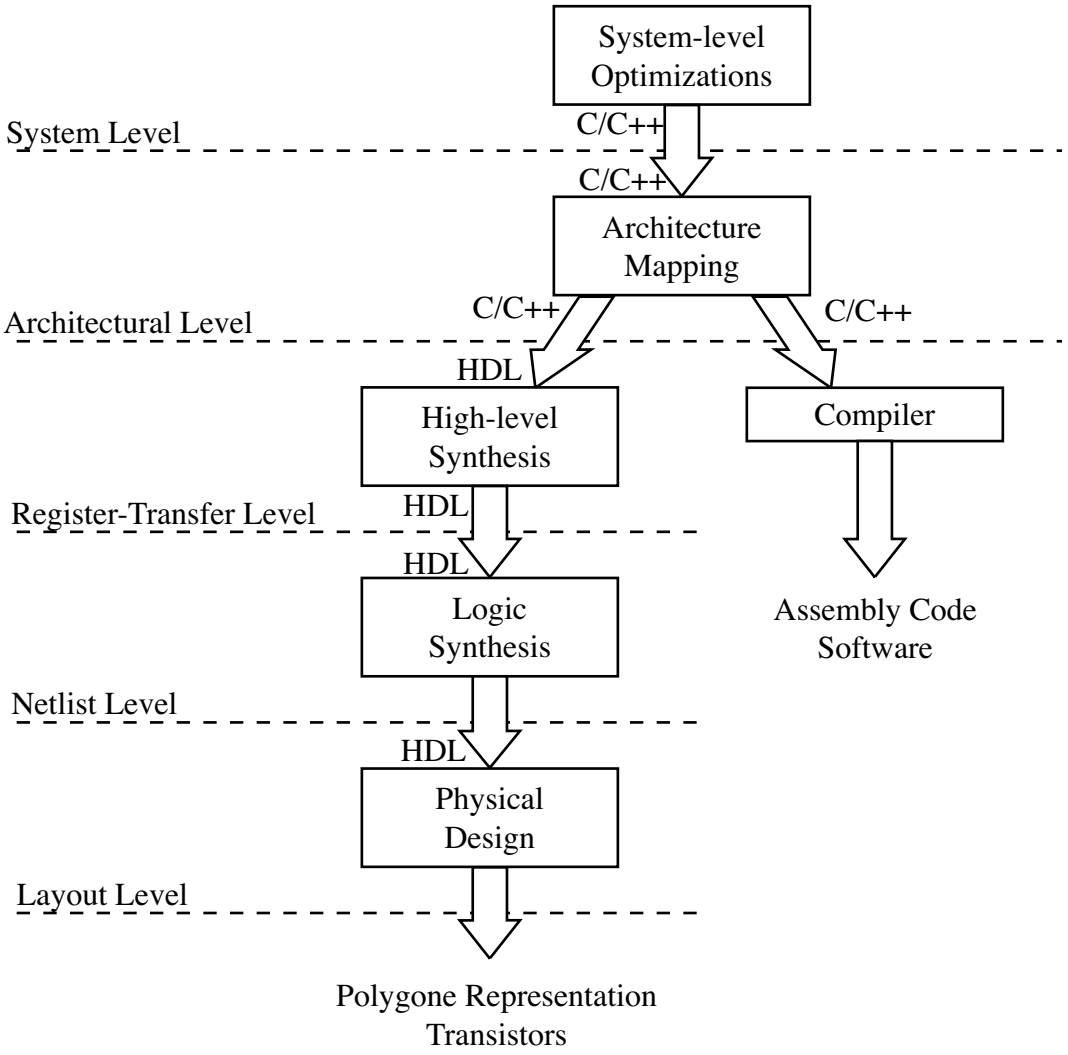


Figure 1.1: top-down design flow of a system

Shared storage is used for inter-process communication whereas internal storage stores local variables and arrays. After architectural mapping and interfaces definitions, the system consists of a set of processes to be mapped to software or hardware. On the software side, compilers can be used to generate assembly code to be executed on a processor. On the hardware side, high-level synthesis may be used to generate a cycle-accurate representation of the design.

High-level synthesis takes a sequential description of an algorithm and automatically implements it as data path and control logic. The data path consists of operators, storage, and steering logic. Operators, which can be shared, usually implement algorithmic or logic operations (e.g. add, subtract, multiply, etc.). Storage is usually implemented as a set of registers or memories. Steering logic represents the connection network between storage and operators. In addition, the control logic implemented as a state-machine or as micro-code controls the internal flow of data (i.e. multiplexer select signals, register update signals, tri-state buffer set signals, etc.). The tasks performed during high-level synthesis consist of scheduling and resource binding according to latency, throughput and resource constraints. Scheduling and binding correspond to the tasks of mapping operations to time slots and operators respectively. The cycle-accurate design generated after high-level synthesis can then be mapped to hardware using a traditional CAD flow consisting of logic synthesis and physical design.

The research presented here fits as a front end to high-level synthesis tools. It can be also seen as a back end to system-level tools performing hardware/software partitioning and interface synthesis. The objective is to take C code instead of HDL as an input to high-level synthesis. From a system-level perspective, the goal is to automate the transition from a architectural description written in C to an input understandable by current high-level synthesis tools.

More specifically, this research in on automating the synthesis of hardware from a C code with such constructs as pointers (including pointer arithmetic and type casting), complex data structures and dynamic memory allocations. These language features are not part of any synthesizable HDL subsets. To make this research possible, the code is assumed

correct and fully visible (complete specification) at compile time. Existing developments both for high-level synthesis and compilers are also leveraged

1.3 Objectives and Contributions

The first goal for this research is to be applicable to a large subset of the C language. As a result, the front-end and initial analysis passes used can deal with any kind of C input without any assumption of a coding style or of a synthesizable subset. Better quality of results may be achieved by restricting the input to a limited subset of C or a given coding style. However, this would defeat our original goal of supporting the full ANSI C. To support all C constructs, my implementation shares the same general-purpose front end and pointer analysis passes as advanced optimizing C compilers. The contribution here is on applying such advanced compiler techniques to the synthesis of hardware from C.

The overall intention of this research is to support full synthesis of C. On the other hand, to make this research possible, existing tools are leveraged when it is possible. As a back end, commercial tools are used to perform high-level synthesis. The flip side of using commercial tools is that they cannot be modified and their internal data structures cannot be accessed in a research environment. In this research, HLS tools are used as black boxes. This leads to some limitations both on the architecture generated and the synthesizable subset. The main limitation is on function calls. First, the implementation of such feature as recursion in general remains a fundamental problem. Tail recursion or limited recursion could be implemented. Without recursion, one way of synthesizing functions is to inline them. However designers often want to map functions to separate components. Today's commercial synthesis tools have many restrictions on functions mapped to components.

Moreover, the synthesis of C functions mapped to components would require some interaction with the HLS tool. For example sharing information would be useful to efficiently synthesize functions with parameters passed by reference. As a result, the synthesis of functions is beyond the scope of this thesis and is left as an Appendix.

The contributions of this thesis are the following.

1) This thesis defines the necessary steps to automate the mapping of ANSI C code into hardware. This work represents the first attempt to efficiently synthesize such constructs as complex data structures, pointers, pointer arithmetic and type casting. The resulting tool bridges the gap between high-level synthesis and system-level tools.
2) A methodology for efficiently supporting dynamic memory allocation and deallocation is also presented. It fits into the general methodology described in Section 1.2 and enables the implementation of recursive data structures into hardware. This thesis also presents how C code with `malloc` and `free` may be automatically mapped to hardware after system-level optimizations (e.g. refining data structures) and architectural mapping (defining internal and shared storage).
3) This thesis also describes a set of techniques to optimize the resulting hardware.
   - Compiler techniques are used to reduced storage before loads and stores. The motivation for these optimizations is to reduce the number of registers necessary in a design.
   - Encoding algorithm is developed to reduce both the bit-width of pointers and the size of the circuits translating pointers' values in assignments and comparisons. Assignments of pointers are especially common at function calls.

- Novel architecture of hardware memory allocator/deallocator is also introduced. The idea is to encode information about the memory block allocated inside of the pointers' value to speed up deallocation.

The outline of the rest of this thesis is the following. In Chapter 2, related work on the different ways of mapping C code onto different target architectures is presented. Chapter 3 is a background chapter in which compiler techniques, namely pointer analysis, and their underlaying memory representations are reviewed in the light of hardware synthesis. The different steps involved in the synthesis of hardware from C code are then presented in Chapter 4. The outcome of this research, presented in Chapter 5, is the tool SpC that automates the translation of C models with pointers, complex data structures, etc. into a code synthesizable by commercially available HLS tools. Results are presented for different application domains. Different optimizations can be applied for efficiently mapping C code onto hardware. In Chapter 6, a set of compiler optimizations to reduce storage before loads and stores are presented. Chapter 7 describes an encoding algorithm that reduces both the bit-width of the pointer variables and the complexity of the circuits implementing comparisons and assignments of pointers. It is important to note at this point that the optimizations presented in Chapters 6 and 7 are also motivated by the synthesis of functions in C as shown in Appendix A. Finally, Chapter 8 presents how memory allocation and deallocation can be optimized by selecting different allocator architectures and by applying compiler optimization techniques. Results are also presented at the end of Chapters 6, 7 and 8 to highlight the benefits of each optimization.

CHAPTER 2. RELATED WORK

C/C++ are two of the most common programming languages. C and C++ are both procedural imperative languages. Their semantics relies on an implicit Von Neuman architecture. In order to be executed, C/C++ programs must be compiled for a given target architecture. C was originally designed to target a generic software architecture consisting of a continuous memory space in which all variables are stored and a microprocessor executing a sequence of instructions. C/C+ code is therefore sequential by definition.

In the last decade, the implementation of compilers evolved significantly to take advantage of new target architectures such as multi-processor systems or very large instruction word (VLIW) machines. At the same time, the features of the language were exploited, for example using object-oriented features in C++, to support new models of computation to model mixed hardware-software systems and higher lever of abstractions. These new constructs may be used to express coarse-grain parallelism, as communicating sequential processes. However finer-grain parallelism at the operation level cannot easily be expressed. High-level synthesis as well as compiler analysis may be used to find such operation-level parallelism.

High-level synthesis (HLS) is defined as the set of transformations necessary to automate the mapping of sequential behavioral descriptions into cycle-accurate descriptions. Current HLS tools input behavioral HDL descriptions. However, HLS tools supporting also C/C++ or C/C++-based languages are starting to appear.

This chapter reviews first related work in the domain of high-level synthesis and modeling using C/C++. A selection of recent works in advanced compilers is then presented, followed by a presentation of current application-specific memory management methodology targeted at mixed software/hardware applications.

2.1 Modeling and synthesizing hardware from C/C++

2.1.1 High-level synthesis

High-level synthesis (aka architectural or behavioral synthesis) consists of generating a structural view of a sequential architectural-level model [14]. The sequential architectural-level model typically consists of a set of parallel processes or tasks. HLS generates a circuit (i.e., mapping to a structural view) consisting of a data path composed of hardware resources and a control unit. The architectural-level model can be abstracted as a set of operations and dependencies. High-level synthesis then performs the following tasks:

- identify the hardware resources (aka *operators*) implementing each operation
- *schedule* the execution time of the operations
- bind these scheduled operations to hardware resources (i.e. *resource allocation*)

In doing so the synthesis tool defines a structural model consisting of a data path, as an interconnection of hardware resources and a cycle-accurate model of a control unit that

issues the control signals to the data path according to the schedule. Resources in the data path represent not only the operators implementing operations in the original model, but also storage (e.g. registers) and steering logic which connects operators, storage elements and Input/Output (I/O) ports.

There have been numerous research projects on high-level synthesis in the past decade [8,10,11,22,23,29,35,36]. Commercial tools are now available. They include Synopsys Behavioral Compiler [91], Mentor Graphics Monet [85], Get2chip Volare [83] and Arexys ArchiMate [72]. These tools take HDL (VHDL or Verilog HDL) as an input. High-level synthesis that take C/C++ as an input can be derived from these tools. Synopsys CoCentric SystemC compiler [92] is such an example.

2.1.2 Hardware description languages and C/C++

C/C++ are software programming languages and have little support for describing hardware efficiently. To model hardware in C/C++, we need the following language features present in HDL but not present in C/C++.

- **Concurrency**: hardware is inherently parallel, while C/C++ programs are inherently sequential. The notion of processes (aka `always` blocks in Verilog HDL), which encapsulates programs that execute concurrently, is introduced. A system is described as a network of processes.
- **Signals**: hardware processes need to use signals (akin to wires or buffered channels) to communicate with one another.
- **Reactivity**: hardware systems are in continuous interaction with their environment, i.e. they are reactive. The notion of reactivity is essential to describing

hardware systems at all levels of abstraction.

- **Data abstraction**: C/C++ supports data abstractions that are useful for software programming. However, for hardware, one needs arbitrary precision signed and unsigned integers, bit vectors and fixed point types.

With such a set of features added, C/C++ can efficiently model hardware/software systems [58]. C/C++ contains many features which are not present in synthesizable subsets of HDLs.

Different subsets of C/C++ and C-like HDLs have been defined and used for synthesis. I mention first those developed in the eighties. HARDWAREC [36] is a language with a C-like syntax and a cycle-based semantics. It models hardware components at the architectural level using C constructs. HARDWAREC can be fully synthesized. However, it makes both restrictions and additions to ANSI C. Constructs are added to define bit-widths, modules and ports. The language includes a quite extended subset of the C constructs including unbounded loops and some form of function calls. Pointers, recursion and dynamic memory allocation are however not supported. CONES [61] from AT&T Bell Laboratories is an automated synthesis system that takes behavioral models written in a C-based language [6] and produces gate-level implementations. Here, the C model describes circuit behavior during each clock cycle of sequential logic. This subset is very restricted and contains neither unbounded loops nor pointers. CONES and HARDWAREC are two examples of C-based HDL found in the literature. However, many other flavors of C-based languages have also been developed and used internally in the industry.

In the recent past, a few projects have been looking at means of using C/C++ as an input to current design flows [15]. 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 [75,81,86]. Even though restrictions on the language apply for synthesis, software/hardware systems can then be modeled directly using C++. Simulation is performed by running the executable generated after compiling the models. Standard debugging environments can then be used to check the functionality of the system.

For reactivity, SYSTEMC [39,86] (formerly known as SCENIC [40] from Synopsys) supports a mixed synchronous and asynchronous approach implemented as a C++ class library. The Esterel C language (ECL) [37] from Cadence is synchronous as it is based on both C and ESTEREL. Other extensions include HANDEL-C [78] and BACHC [30] originally based on OCCAM, SPECC [89] loosely based on SPECCHART, OCAPI from IMEC [81] and CYNLIB from CynApps [75]. The main initiatives to standardize such extensions are the Open SystemC Initiative [86], the Accelera Working Group [71] (formerly, OVI/VI), the SpecC Technology Open Consortium [89], and the recent DATC++ technical subcommittee of the IEEE Computer Society Design Automation Technical Committee (DATC) [79].

2.1.3 Hardware synthesis from C/C++

In order to map functionality to hardware, a synthesizable-C/C++ subset is usually defined. We can distinguish two approaches. The first method 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 high-level 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 [74], OCAPI [54,81], CYNAPPS [75] and others [76,94] automated the translation from a refined RTL model to HDL. These subsets do not include pointers.

Two commercial tools, C-Level Design System Compiler (formerly Compilogic C2HDL) [73] and Frontier Design AR|T BUILDER [77], also provide tools for translating C models into Verilog or VHDL. Limited scheduling and resource-sharing techniques can be applied to 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. System Compiler on the other hand, supports all of the ANSI C constructs excluding libraries. However, pointers are implemented in a software-like approach. In the general case, they are considered as addresses to data stored in memories, which requires the allocation of memories, to store the various variables, and addressing units. In hardware, designers may want to optimize locality by mapping data to multiple memories, registers or even wires (e.g. output of functional units) in the physical implementation.

Kim and Choi [34] as well as the author of this thesis [55,56,57,59] 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.

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 [25,92] and by NEC with Cyber [63,64]. As we have seen in Section 2.1.1, high-level synthesis enables the mapping of sequential functional descriptions onto hardware. Synthesis from C/C++ description can leverage some of this previous work 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 is also 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++ model mix hardware-software and other specific architectures. For hardware-software codesign, the CoWare N2C system [74] 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 [20] uses C*, another superset of C with processes and timing constructs. During hardware synthesis, functions are inlined and pointers are only treated 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 [43], a bottom-up methodology is presented in which a library of components is 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 [26]. For mixed software and reprogrammable FPGA architectures, the GARP compiler [7] as well as the NIMBLE compiler [38] automatically generate retargetable co-processors to speed up loops. Pointers are treated as references to the main memory. This approach is relevant for implementing memory-mapped I/O. However, it can be a limitation to parallelize data transfers inside of a data path. Finally, Babb et al. [2] present a compiler for a variation of the RAW 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 as these locations are mapped to a unique memory and therefore cannot be accessed in parallel in a data path.

To summarize the previous work on the synthesis of hardware from C, pointers and dynamic memory allocations are two of the main outstanding issues for the synthesis of hardware from C. In order to guarantee a 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. In a sense, the problem of mapping C code onto hardware can be compared with compiling C code onto distributed systems or systems in which parallelism is explicit (e.g. VLIW processor). The approach presented in this thesis is based on the use of advanced compiler-analysis techniques to efficiently map such C constructs as dynamic memory allocation and pointers onto hardware.

2.2 Software compilation of C and C++ onto parallel architectures

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, 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 managing the memory. By definition, pointers may reference multiple data. Such case happens when referencing the different elements of a data structure or of 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 [1,45]. 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 Chapter 6 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 depends

on the data accesses at each statement in the program. To model the effect of statements involving pointers, 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 [41]. 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 [50,60,66,67] 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.

2.3 Application-specific memory management methodology

For decades, memory management has been one of the major development areas both for software and computer architecture. In software, at the user level, memory management is typically performed by the operating system.

In hardware, memory bandwidth is often a bottleneck in applications such as networking, signal processing, graphics and encryption. Memory architecture exploration and efficient memory management technology are key to the design of new high-performance systems. Memory generators commercially available today [88] enable fast integration of memories in a system. Scheduling of memory accesses has also been integrated into most commercial HLS tools. Most of the refinement and compilation steps developed for soft-

ware applications can also be used for hardware. Nevertheless, a software methodology usually assumes a fixed memory architecture, which may be general purpose or application specific like in a DSP or ASIP. In hardware, at the behavioral level, designers would typically explore different memory architectures in order to trade-off area and power for a given timing constraint.

A methodology for the design of custom memory systems has been described by Catthoor et al. [9]. It is defined for two sets of applications, networking and signal processing, and supports a limited subset of C/C++. The basic concepts presented in Catthoor's work can be generalized to support a larger subset of the C syntax for an extended set of applications. Two main steps can be distinguished in the methodology: we describe briefly here the transformations performed first at the system level, and then at the architectural level.

At the system level, the functionality of the algorithm is verified. Data formats are refined. For example, after quantization, the format of data can be refined from floating-point to fixed-point [31]. Data structures can also be refined for example to reduce the number of indirect memory references. Examples of such transformations for networking applications have been studied by Wuytack et al. [69].

At the architectural level, after partitioning, the system typically consists of multiple communicating processes to be mapped to hardware or software [25,58,74]. Memory segments are defined for internal storage and/or shared memory. These memory segments can then be mapped to one or multiple memories during synthesis. Some of the storage area (e.g. internal variables, etc.) can be statically allocated during synthesis or compilation. However, to support dynamic storage allocation (e.g. for recursive data structures), allocation and deallocation primitives implemented in software or hardware shall be defined.

In software, memory allocation and deallocation are implemented as primitives that are part of the operating system (OS). These primitives can be called in a C program using the functions defined in the standard library (e.g. `malloc`, `free`, etc.). Many schemes have been developed for OS to manage memory. An extensive survey by Wilson et al. [65] presents many of the techniques used for memory allocation and deallocation in software.

Memory management can also be implemented in hardware. For memory allocation and deallocation, instead of the system calls to the OS, requests are sent through signals to an *allocator* block (aka. *virtual memory manager* [70]) implemented in hardware. Its interface is shown on Figure 2.1. Internally, the allocator stores a list of the free blocks in memory as well as a list of the allocated blocks. To allocate memory, the size of the block to be allocated (*malloc_size*) is sent. The allocator then searches in its free list a big enough block and returns the address of the beginning of this block (*malloc_address*). Two techniques are often used: *first fit* where the first acceptable free block is returned or *best fit* where the block of minimal size is returned. To free previously allocated memory, the address of the block to be deallocated (*free_address*) is sent to the allocator. The allocator then searches inside of the allocated list the block and adds it back to the free list. Adjacent free blocks can then be merged. An optimized architecture to speed up memory allocation in hardware is presented by Chang et al [12]. The implementation itself of the

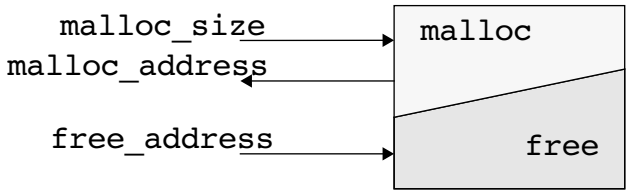


Figure 2.1: Interface of the allocator block implementing `malloc` and `free`

allocator may also vary according to the application and the data structures. A number of these application specific implementations are presented by Wuytack et al.[70].

Once an architecture is decided, hardware can be implemented using synthesis tools and compilers can be used for software.

As far as memory management is concerned, scheduling of memory accesses, register/ memory allocation and address generation can be integrated into synthesis tools and compilers. In current commercial synthesis tools, each array is manually mapped to register files or memories of different types. For scheduling, the characteristics of the memories (number of ports, read/write latency, etc.) are defined as part of a components library. Recent research works have been looking at techniques to automatically perform memory assignment, address generation and optimized scheduling according to the memory type. The latest developments of these techniques have been presented by Catthoor et al. [9] and Panda et al. [47].

CHAPTER 3. BACKGROUND

In software, a C program is targeted to a virtual architecture consisting of one memory space in which all data are stored. The semantics of pointers is the address of an element in memory space. Even though `register` declaration may allow programmers to specify the variables to place in registers, the assignment of variables to registers is generally done by the compiler. The notions of caches and memory pages are transparent to programmers.

In hardware, at the behavioral level, designers want to have control on where data are stored and want to optimize the locality of the storage. Typically, a chip design contains multiple memory banks, register files, registers and wires. To efficiently map C code onto hardware, the storage space must be first partitioned into distinct memory blocks. The task of the synthesis tool is then to find an efficient physical implementation (memory, register, wire or ports) for these memory blocks. Some of these blocks may also represent pointers. Pointers may be used to reference any variable no matter where its information is available (i.e. no matter what its physical implementation is). Pointers are then considered as references: references to memory elements, registers, wires or ports. In particular, pointers can be used to allocate, read, write and deallocate data. In this thesis we call the action of read-

ing data using a pointer a *load*. Subsequently, a *store* is the action of writing data using a pointer. Allocation and deallocation are performed through the standard library functions `malloc` and `free`.

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

Example 3.1. *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`*,* `in.b` *in separate registers 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:*

```
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:*

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

Example 3.2. *Let us consider an application, where a hardware block receives objects of different sizes and processes them. In the final implementation, after partitioning the memory space, some of the intermediate data are stored in registers or memories. In this example, some of the objects received are copied into a register (*`reg`*). Some other are only used within this block and are stored in a private memory (*`local_RAM`*). Finally some, larger, may also be accessed by other blocks and are stored in a shared memory (*`shared_RAM`*).*

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

```
  p = malloc(8);
...
// store in reg, local_RAM or shared_RAM
*p = object.data;
...
if(object.type != REG)
  // free storage in local_RAM or in shared_RAM
  free(p);
```

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

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

As we can see in Examples 3.1 and 3.2, in order to efficiently map C code into hardware, we first need to partition the memory space into blocks that may be mapped to memories, registers or wires in the final implementation. In our implementation, memory space is represented as a set of location sets, described in Section 3.1.

Subsequently, to synthesize *loads*, *stores* and `free` operations into hardware, we need to know at compile time the set of locations the pointers may reference (*points-to* information). Such information is also widely used in compilers. In order to parallelize programs onto distributed architectures, the independent sets of data, which can be processed in parallel, have to be extracted. The problem there is to find statements in the program that may read or write the same locations (aliasing problem). For this purpose, the *aliasing* informa-

tion has to be determined between pointers. The points-to information and the aliasing information are equivalent and can be determined by recent analysis techniques called *pointer analysis* or *alias analysis* described in Section 3.2.

3.1 Memory space 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. As a result, most analysis techniques combine elements within a single data structure. Some techniques combine elements based on their allocation contexts [66,67] or on limiting the length of access paths to some fixed constant (*k-limiting*). Shape analysis [17,24] gives the most accurate representation as they may distinguish trees from DAGs, linear lists from cyclic lists and so on. However its implementation to support large C programs remains challenging.

In order to find both an accurate and a practical representation for hardware synthesis, we use the notion of *location sets* introduced by Wilson and Lam [66,67]. 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

$\{f + i.s \mid i \in \mathbb{Z}\}$ in a particular block of memory *loc*. That is *f* is an offset within a block and *s* is the stride. If the stride is zero, the location set contains a single element. Otherwise, it is assumed to be an unbounded set of locations. Table 3.1 shows the location sets for various expressions.

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

Table 3.1: Location set examples (*f*=offset of field F), (*s*=stride or array element size)

For simple data structures (arrays, structures, array of structures), offsets are used to identify the different fields of structures whereas strides are used to record array-element sizes. Figure 3.1 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 scalar variables and structures. This makes sense since all elements of an array are usually alike.

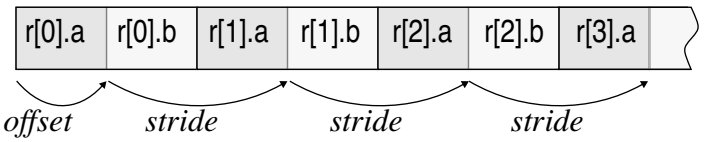


Figure 3.1: Representation of `struct{int a; int b;} r[]`; the offset and stride correspond to the locations `r[i].b` where `i` is integer.

Nested arrays and structures, type casting and pointer arithmetic make things more complicated, leading to some additional inaccuracies. Example 3.3 shows how references to array nested in structures are represented approximately. The array bound information

in the declared type cannot be used because the C language does not provide array-bounds checking. A reference to an array nested in a structure could access other fields of the structure by using out-of-bound array indices.

Example 3.3. *Consider the array* `r.F[]` *nested in a structure* `r`*:*

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

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

Dynamically-allocated memory locations (aka. heap-allocated objects) are represented by a specific location set. As far as accuracy, it would not be practical to distinguish every element of a recursive data structure. Therefore, the goal of this representation is to distinguish complete data structures. The different elements of a recursive data structure would typically be combined into one location set. For example, we want to distinguish one list from another but we do not want to distinguish the different elements of a list. Heuristics are used to distinguish dynamically allocated data. Storage allocated in the same context is assumed to be part of the same equivalence class. Within one function, storage allocated by a given `malloc` in the code is represented by a unique location set. When `malloc` is called within a function, a different location set is created for each call chain (context). These heuristics have been proven to work well as long as the program uses the standard memory allocation routines [66].

Example 3.4. *In the code segment shown in Example 3.2, the memory space can be represented by the following set of location sets:* <p,0,0>, <reg,0,0>, <object,0,0> *for* `object.type`, <object,4,0> *for* `object.data`, <object,8,0> *for* `object.data2`, <malloc1,0,0> *for the storage allocated by the first malloc call (*`malloc(4)`*), and* <malloc2,0,0> *for the storage allocated by the second malloc call (*`malloc(8)`*).*

3.2 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 the pointer may point to. With the memory representation of Section 3.1, 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 main types of analyses (there are also flow-insensitive and context-sensitive analyses).

1) *Flow- and context-insensitive:* the analysis [60] 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 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 [66,67] by Wilson and Lam, within the SUIF framework, can efficiently support the full-featured ANSI C with good accuracy for hardware synthesis. Even though the complexity of the analysis can be exponential, it is not a limitation because hardware models are rather small and simple programs.

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.

The implementation presented in this thesis uses a flow- and context-sensitive analysis. Note that context-sensitivity is useful for synthesizing both functions mapped to components (cf. Appendix A) and for pointers to functions (cf. Section 4.2.2.2). 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 3.5. *In the code segment presented in Example 3.1, annotations are inserted by the pointer analysis to specify what pointers may point to (i.e. points-to set) 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 `<p,0,0> -> {<in,0,0>,<in,4,0>}` *stands for "*`p` *may point to variables* `in.a` *or* `in.b`*" where* `in.a` *is represented by the points-to set* `<in,0,0>` *and* `in.b` *is represented by the points-to set* `<in,4,0>`.

Example 3.6. *In the code segment presented in Example 3.2, annotations are inserted by the pointer analysis to specify what pointers may point to at loads, stores and* `free` *calls.*

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

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

3.3 Definition of the subset

This section is only about the restrictions on the synthesizable subset. Limitations on the generated architecture may also exist akin to the limitations of the behavioral synthesis

tool used as a back end. In particular, the techniques presented here do not depend on the type of memories (SRAM, DRAM, etc.).

The pointer analyses and memory representation presented in the previous sections support the complete ANSI C syntax. In this thesis however, a synthesizable subset is defined. This subset includes `malloc`/`free` as well as all types of pointers and type casting. The code is assumed to be correct. Tools such as Purify [87] or LCLint [21,84] can be used to find memory leaks and check memory reads, writes and deallocations. Besides, the following two restrictions are defined.

The first restriction applies to systems described as a set of parallel processes: pointers that reference data outside of the scope of a process (e.g. global variables or data internal to some other processes) are not allowed. Their resolution would require the synthesis of some kind of interface between the circuits realizing the processes. As presented in Section 1.2, such interface is usually defined during system partitioning and, hence, before synthesis. As a result, memory allocated in one process is assumed to be accessed and deallocated only within this same process.

The second limitation stems from the fact that most commercial synthesis tools also have restrictions on functions. Recursion is usually not supported as its implementation in hardware in general remains a fundamental problem. This is not a real issue in reality as recursion is often not well suited for hardware implementation. In theory, tail recursion could be supported as it can be transformed into a loop. Limited recursion could also be supported by modeling a stack using the techniques presented here for dynamic memory allocation.

During synthesis, functions may either be inlined or mapped to components. Apart from recursion, there is no real restriction on functions that are inlined. On the other hand, 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 therefore the resolution of pointers and `malloc`/`free` inside of functions, is beyond the scope of this thesis. Appendix A gives an outline of how functions can be synthesized.

CHAPTER 4. SYNTHESIS OF HARDWARE FROM C

This chapter presents the different steps involved in the synthesis of hardware from C. First, the memory space is partitioned into a set of location sets which are physically implemented as wires, registers, or memories. Some of these location sets represent pointers. Pointers are resolved by encoding the value of the pointers and creating branching statements for loads and stores. Finally dynamic memory allocation and deallocation are performed by custom hardware memory allocators.

A prototype of tool implementing these techniques is presented in Chapter 5. The implementation can be further optimized. In Chapter 6, compiler techniques are used to optimize the storage before loads and stores. An algorithm to efficiently encode the value of the pointers is presented in Chapter 7. Finally, in Chapter 8, a library of hardware memory allocators are defined to optimize the implementation of `malloc` and `free`.

4.1 Memory Refinement

After analysis, the storage in the program can be represented as a set of distinct locations sets. This set of location sets represents a partitioning of the memory space. Each location set is ultimately mapped to a wire, a register or a section of memory in the final design as shown on Figure 4.1. The allocation of a given scalar variable to a register or a wire as well as the mapping to memories or register files are typically the result of high-level synthesis (HLS). In this section, we present how distinct location sets can be mapped to a set of arrays and variables. We do not consider pointers and heap objects. The synthesis of pointers and `malloc`/`free` is presented in Sections 4.2 and 4.3. In the rest of the thesis, we use the following representation for *fundamental* (or basic) types: `char` and `unsigned char` are represented as 8 bits, `short` and `unsigned short` are represented as 16 bits, and `int` and `unsigned int` are represented as 32 bits. These representations are the most common on 32 bits architecture. Derived types such as pointers, arrays and structures are constructed from these fundamental data types.

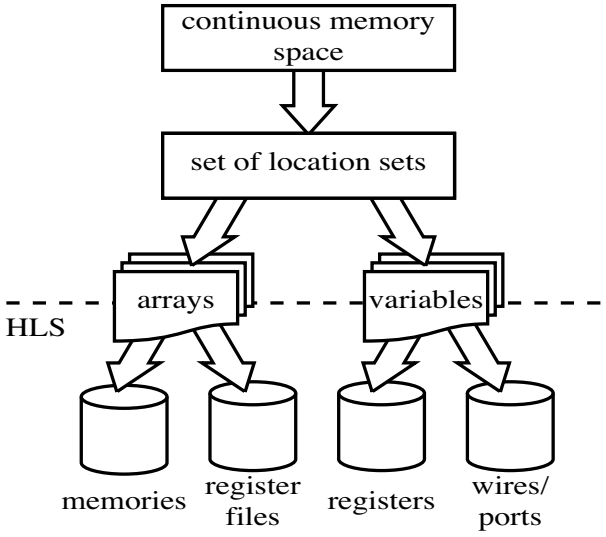


Figure 4.1: Memory refinement from a continuous memory space to a set of memories, registers and wires

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 non-zero strides (i.e. sets of multiple locations). A singleton location set may therefore be treated as a simple scalar variable, whereas a location set with non-zero stride may be mapped to an array. In our implementation, for each location set <*loc*, *f*, *s*>, we define `SPC_`*loc_f_s* as follows.

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

For a location set with non-zero stride (i.e. *s* not null), `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 [9,47] during high-level synthesis. For arrays of structures, the different fields of the structures can be mapped to different memories as long as their representations do not overlap. The different fields of the structures can then be independently accessed, leading to more flexibility and potentially better performances.

Example 4.1. *Consider the following structure variable.*

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

Four location sets represent the four fields of the structure `csi`. *On the specific target architecture, the fields* `csi.c1`, `csi.c2`, `csi.s` *and* `csi.i` *are respectively represented by the location sets* <`csi`, *0, 0*>, <`csi`, *1, 0*>, <`csi`, *2, 0*> *and* <`csi`, *4, 0*>. *The layout in memory space before synthesis is represented on Figure 4.2.*

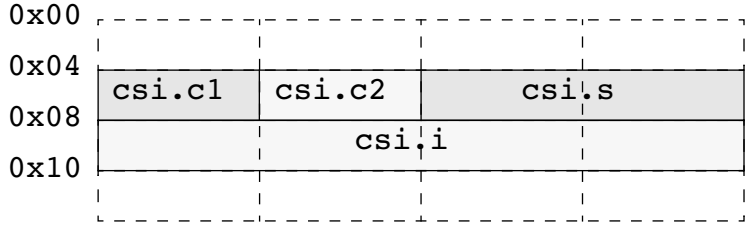


Figure 4.2: Memory layout of `struct {char c1; char c2; short s; int i;} csi.`

We create the following variables corresponding to each location set:

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

As a result during the mapping to hardware the assignment

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

is replaced by

```
SPC_csi_1_0 = 0;
```

Out of bound array accesses, as well as copies of structures can make things more complicated. With our memory representation, one data (e.g. an entire structure) may be represented by the concatenation of multiple elements of location sets. In Example 4.2, a structure is represented as two integers. In Example 4.3, an integer inside of a structure is represented by the concatenation of two short integers.

Example 4.2. *This example illustrates the implementation of a structure copy.*

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

After translation, the following synthesizable code is generated:

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

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

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

Example 4.3. *In the following code segment, the structure variable* `its` *contains an array of short integers.*

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

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

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

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

// its.i = a;
SPC_its_0_2[0] = SPC_a_0_0 >> 16;
SPC_its_0_2[1] = SPC_a_0_0 & 0xffff;

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

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

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

4.2 Pointer Synthesis

In hardware, as discussed in Chapter 3, data may be physically available in 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, each pointer may point to. The synthesis tool generates the appropriate circuit to dynamically access these locations according to the pointers' value. Two types of pointers may be defined: 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 non-zero 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, the variable name `p` is used as a generic pointer name. We have also seen in the previous section how complex data structures can be mapped to arrays and scalar variables. Without loss of generality, pointers are considered either as scalar variables or arrays that may point to variables or arrays.

4.2.1 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 $\lceil \log_2(\texttt{size\_of\_points-to-set}) \rceil$. The index `p.index` on the other hand is used when the pointer `p` may point to a location set with non-zero 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 scalar pointer variables, these two fields can be implemented as separate variables `p_tag` and `p_index`.

Definition 4.1. *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 on Figure 4.3. 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 32bits). 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 even after type-casting.

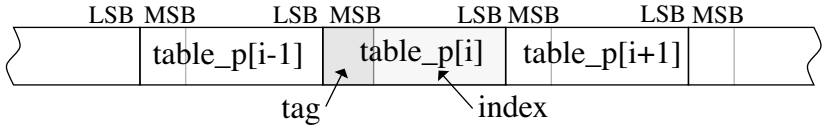


Figure 4.3: Encoding of pointers in an array

Example 4.4. *Consider the following code segment.*

```
int *p, *q;
int a, b, c, table[256];
...
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 one-bit 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 4.5. *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. Implementing pointers' comparisons and assignments efficiently requires 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 4.5 gives two examples of encodings that produce different implementations for the assignment of two pointers. An encoding algorithm that reduces both the bit width of the pointers and the size of the circuits translating the values of the pointers is presented in Chapter 7.

4.2.2 Dereferencing the pointers

Pointers may point to multiple locations. As a result, for every load and store, a circuit has to be created to dynamically access the different locations the pointer may reference at this point in the program. This circuit can be described as a set of branching statements in which the locations in the points-to set are directly accessed. *Dereferencing pointers* corresponds to the task of replacing the loads and stores in the program by these branching statements.

Several types of pointers can be distinguished. We have seen in Section 4.1 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 pointer to function.

4.2.2.1 Pointers to variables and arrays

We use the result of pointer analysis to remove loads and stores. With the assumptions of Section 2.3, 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 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 4.6. *Consider the code segment in Example 4.4,* `*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 pointer* `p` *points to (i.e.* `*p`*) at the load instruction. Similarly for the store we create the temporary variable* `tmp_q` *that stores the new value to be assigned to the data* `q` *points to at the store instruction. After encoding the pointers' value (cf. Example 4.4) 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;
}
```

The corresponding circuit generated after synthesis is presented in Figure 4.4. Note that the load (`...=*p`*) is implemented by a 2-input multiplexer controlled by* `p_tag`.

The removal of the dereferences '`*`' in loads and stores can be done in one pass. For each load `(...=*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 4.6) 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=...)`, 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 an assignment to this location. Otherwise we create a temporary variable (`tmp_q` in Example 4.6) that stores the value to be assigned to the data `q` points to. The store is then

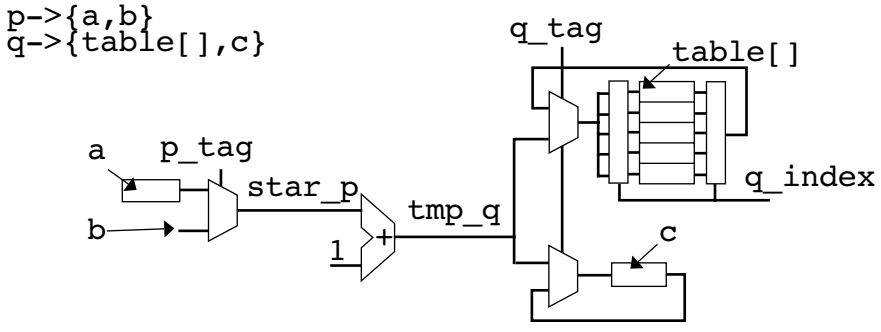


Figure 4.4: Implementation of `*q=*p+1`, where `p` may point to `a` or `b` and `q` may point to `c` or an element of `table[]`.

replaced by an assignment to this temporary variable and branching statements are inserted after the store to update the values of the variables `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 6, we also present some optimizations to reduce the memory usage before loads and between loads and stores when the pointer is a variable.

4.2.2.2 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.

Pointers to pointers

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

Example 4.7. *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 follows using a sequence of two case statements. For the sake of clarity the pointers' values have not been encoded. Encoding of the pointers' 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
    case &q1:                       //       further analysis
        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 4.8. *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;
}
```

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 this thesis, functions are inlined before synthesis.

4.3 Synthesis of `malloc` and `free`

In order to support dynamic memory allocation and deallocation, the hardware needs to access an allocator. In general the allocator could be implemented in software (for mixed hardware/software implementations) or completely in hardware. For example, in a mixed hardware/software implementation, the hardware can send an interrupt to the processor to perform memory allocation and deallocation. In the case of an allocation, the software, typically the operating system running on the processor, would then send the address of the allocated block in memory to the hardware. Since this work is on the hardware synthesis of C code, only a hardware implementation is presented. Nevertheless, the techniques presented here could also be targeted to a software implementation.

In software, `malloc` and `free` are implemented as standard library functions. Similarly, for hardware synthesis, we use a library of hardware components implementing `malloc` and `free`. The idea here is to have one component, called *allocator*, implementing both the `malloc` and `free` functions as introduced in Section 2.3. In order to efficiently manage memory, multiple customized memory allocators that may allocate storage in multiple memories in parallel shall be supported. As a result, the memory space in which data are dynamically allocated (heap space) is partitioned into a set of *memory segments* (pools of blocks).

Definition 4.2. *A* memory segment *is defined as an array of finite size in which data are allocated by a unique allocator. This array may later on be mapped to one or more memories during high-level synthesis.*

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

example, tools such as the one defined in the Matisse research project [69,70,80] could be used in order to refine data structures and define different arrangements and architectures of hardware memory allocators.

For each `malloc` in the code, the designer selects in which memory segment the storage is allocated. Since the size of the dynamically allocated memory cannot be found by static analysis, the designer also sets the size of each memory segment manually. The tool instantiates then the hardware memory allocators corresponding to each memory segment and synthesizes the appropriate circuit to allocate, access and deallocate data.

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

For a call `free(p)`, the data to be deallocated may be in one memory segment or another depending on the value of the pointer `p`. We generate branching statements in which the different allocators corresponding to the different memory segments may dynamically be called according to the pointer's *tag*. The pointer's *index* is then sent to the allocator to indicate which block should be deallocated. Loads, stores and addresses are resolved as shown in the previous section. Examples 4.9 and 4.10 illustrate how `malloc` and `free` calls are resolved while removing pointers.

Example 4.9. *Consider the following code segment.*

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

If `malloc` *is mapped to a memory segment called* seg1 *of size 32 bytes, we generate the following code (assuming that the size of* `char` *is one byte):*

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

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

Example 4.10. *Let us now consider a more complex example where pointer* `p` *may point to different memory segments.*

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

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

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

If each memory segment is mapped to a different RAM during synthesis, we end up with the architecture on Figure 4.5.

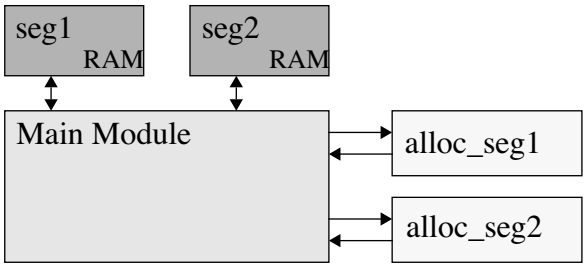


Figure 4.5: Architecture for multiple memories and allocators

4.4 Summary

The general methodology for efficiently mapping C code from a system-level description onto hardware and generating an application-specific memory management architecture are respectively presented in Sections 1.2 and 2.3. In this chapter, the techniques for resolving pointers, `malloc`/`free` and complex data structures within this context are presented.

After analysis, the memory space is partitioned into a set of location sets which may physically be map to memories, registers or wires/ports during synthesis. Pointer analysis is used to define the set of location sets each pointer may point to.

This points-to information is used to replace loads and stores in the code by branching statements in which the data referenced are dynamically accessed. Values of pointers are encoded. In this encoding, the most significant bits correspond to the *tag*, which encodes the location set referenced. The remaining bits (least significant bits) correspond to the *index*, representing which location within a given location set is actually referenced.

`Malloc` and `free` calls are replaced by calls to a specific allocator function mapped to a hardware allocator component during synthesis. For each `malloc`, memory allocation is performed within a *memory segment* defined by the designer (or by higher level tools). Each memory segment has a fixed size and is managed by a unique allocator. Deallocation may be apply in different memory segments. Branching statements are inserted to dynamically free previously allocated memory within these memory segments according to the value of the pointer.

CHAPTER 5. IMPLEMENTATION

The different techniques presented in the previous chapter have been implemented in the tool *SpC* (for *Synthesis of Pointers in C*) which automates the synthesis of a C architectural description into hardware. The toolflow is presented here, followed by some results for a set of applications.

5.1 Toolflow

In the methodology described in Section 1.2, the design of a system starts at the system level, where data structures, data formats and algorithms are refined. At the architectural level, the system consists of a set of communicating processes. Each of these processes can then be mapped to software or hardware. The general tool flow is shown on Figure 5.1. Our tool SpC inputs a C function after architectural refinement (i.e. communication and internal/shared storage are defined). This C function may contain pointers.

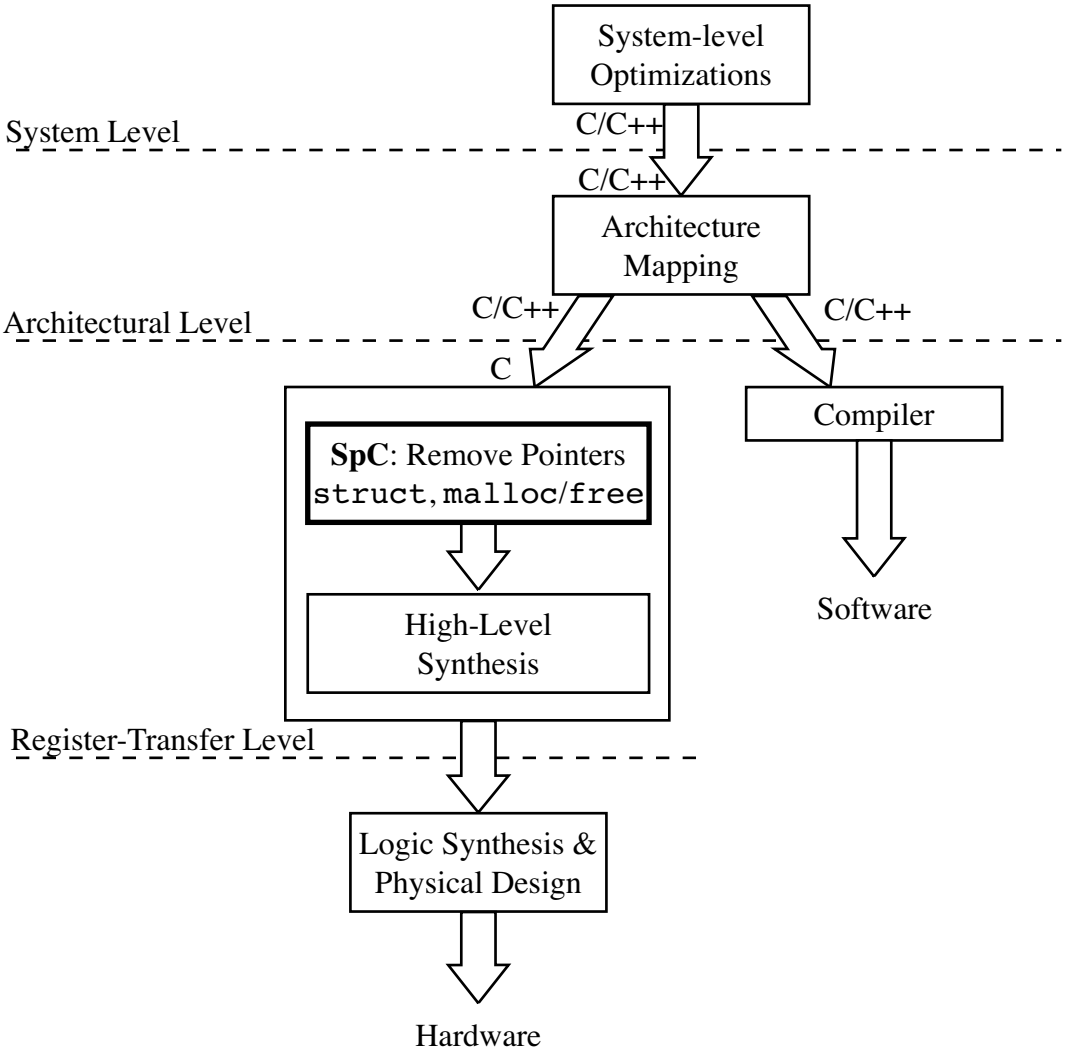


Figure 5.1: Position of SpC between high-level synthesis and architectural mapping

The different techniques presented in Section 4 have been implemented using the SUIF compiler environment [68,90]. The tool SpC consists of the following passes.

1) The compiler front end takes a C function and translates it into a SUIF intermediary representation (IR).
2) The points-to sets of each pointer in the code is defined by means of pointer analysis. A flow and context-sensitive implementation by Wilson and Lam [66,67] is used.

3) Calls to `malloc` and `free` are replaced by calls to specific allocation functions.

4) Pointers are resolved. Loads and stores are replaced by branching statements in which the different locations the pointer may reference are dynamically accessed. Pointers' values are encoded.

5) Memory space is partitioned. Each location set is mapped to an array or a scalar variable. Accesses to the different location sets are replaced by accesses to their corresponding arrays or scalar variables.

6) C code without pointers, `struct` or `malloc`/`free` is generated back from the IR.

The previous passes represent the steps implemented inside SpC. The tool itself can be used within two different toolflows as shown on Figure 5.2. First, SpC may be used within a traditional C-to-HDL methodology as shown on the right part of Figure 5.2. The C code output of SpC is very much like an HDL model. It can be automatically translated to Verilog (using syntax tree mapping from C IR to Verilog IR). The calls to specific memory allocation functions are handled as follows. During the translation into HDL, the different allocators corresponding to each memory segment are instantiated and the specific allocation functions are mapped to these allocator modules. The communication between each allocator and the main module is done using hand-shakes. The resulting Verilog module can then be synthesized using Synopsys Behavioral Compiler [91].

Another flow integrates SpC within the recent Synopsys Cocentric SystemC Compiler environment [92]. The idea here is to take a SystemC description as an input instead of a simple C function. In this SystemC description, the processes to be mapped to hardware

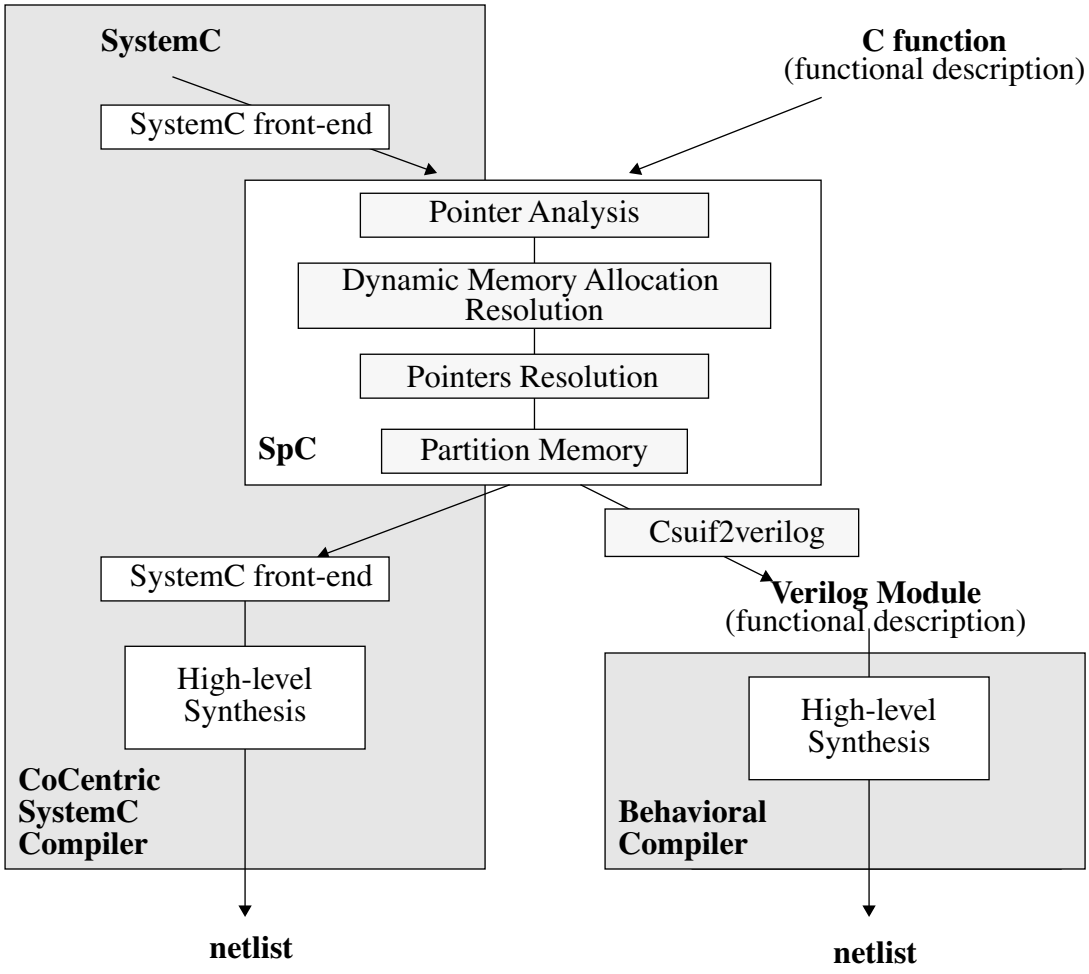


Figure 5.2: SystemC and a traditional C-to-HDL methodology using SpC

are implemented by functions called using a call-back mechanism [86]. For functions containing pointers, a pragma can be used to specify that pointers have to be removed prior to synthesis. These functions are then exported. During this process, the i/o ports of the enclosing SystemC module are transformed into global variables. Here, although SystemC is based on C++, the assumption is that the function body uses only C syntax. The function exported is then parsed within the SUIF compiler framework and pointers are removed using SpC. The resulting C function without pointers is then reintegrated within the Sys-

temC environment. The initial call-back function with pointers is simply replaced by the newly generated function without pointers which can be synthesized.

5.2 Results

We show the results for the resolution of pointers in relatively large 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 real examples of C descriptions to logic gates. For multimedia applications, SpC was used for the implementation of a two-dimensional inverse discrete cosine transform (2D IDCT) [42], an alpha blender and a video image filter. It was also used to generate the specific-purpose and general-purpose memory allocators, which will be described in Chapter 8. Finally as an example of networking application, an ATM segmentation engine was also synthesized to hardware. The multimedia applications are described first.

In multimedia applications, it is often hard to find relevant examples involving pointers to multiple locations and complex data structures without function calls. Pointers are often used to scan arrays within loops using pointer arithmetic. These pointers are relatively straightforward to remove using constant propagation and standard compiler loop optimizations such as loop unrolling. Our examples contain pointers which cannot be removed using such techniques.

The 2D IDCT is widely used in image compression standards such as JPEG, MPEG and H263. The 2D IDCT implemented consists of two one-dimensional IDCTs (1D IDCTs). For this purpose, we use three different memories: the input buffer (`in_table`), the intermediate buffer that stores the result of the first 1D IDCT (`buf_table`) and the

output buffer (`out_table`). These memories are accessed through pointers and pointer arithmetic. Pointers are also used in the 1D IDCT to reference two register banks (`buff1` and `buff2`).

The 2D IDCT is implemented using only one call to 1D 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
  }
}
```

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 2D IDCT algorithm in which the `1d_idct` function is called twice would lead to two 1D 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 8x8. 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 oth-

ers (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 third example is a filter used in the JPEG library of Synopsys COSSAP [93] and is used, for example, for RGB to YCrCb transformations. The filter implements the operation $\boldsymbol{Y}[i] = clip(\boldsymbol{A} \cdot \boldsymbol{X}[i] + B, C)$ for $i = \{1, 2, ..., n\}$, where $\boldsymbol{A}$ is a 3×3 matrix, B and C are vectors and $\boldsymbol{Y}$ and $\boldsymbol{X}$ are two $3 \times n$ dynamically-allocated matrices.

The next example is the implementation of an ATM segmentation engine. The segmentation engine receives frames to be sent from the host. These frames are segmented into 48 byte cells (payload of an ATM cell) to be transmitted on the network. The engine keeps track of each frame in a queue. For every new frame, a new virtual connection is opened and a new queue element is allocated. As a results, we have two sets of `malloc` calls: one to allocate queue elements, the other to allocate connection status records. Finally the last two examples correspond to the specific-purpose and general-purpose allocators to be introduced in Chapter 8. They implement both `malloc` and `free` in hardware.

The results after synthesis are presented on Table 5.1. The first column gives the name of the example described above. Columns 2 and 3 respectively show the number of lines of C code and of Verilog code automatically produced. The CPU time for translating the C code into HDL using SpC on a Sun Ultra2 is shown in Column 4. The results highlight that SpC can deal with medium-size programs is a very short period of time. The Verilog modules are then synthesized by Synopsys Behavior Compiler. Results after synthesis are given. Columns 5 and 6 present the area of combinational and non-combinational (i.e. reg-

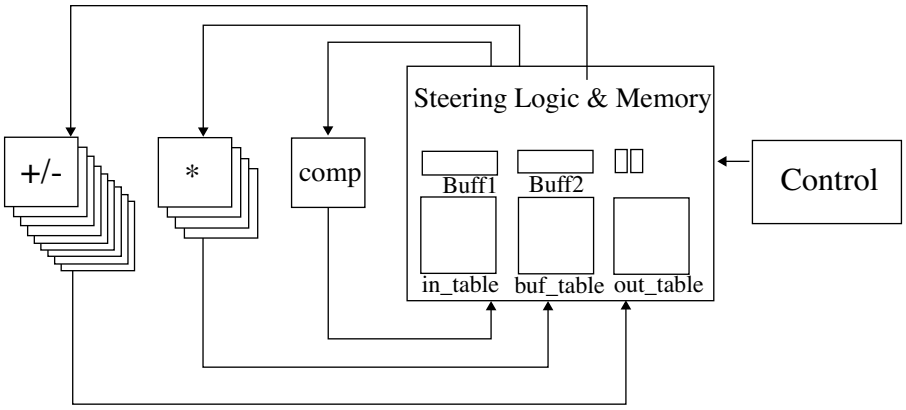


Figure 5.3: architecture of the 2D IDCT

ister) logic. The clock frequency shown in the last column is 100MHz except for the video filter example.

These results show that high-performance implementations can be generated from C programs. As an example, the implementation of the 2D-IDCT is presented in Figure 5.3. The design consists of five multipliers and nine adders or subtracters. This 2D-IDCT was designed to sustain the MPEG-2 bit rate with less than 400 clock cycles to perform a 2D-IDCT on an 8x8 block. Other implementations can be found by changing the timing and resource constraints.

test	C lines	Verilog lines	cpu time (in s)	area (x1000)		frequency (MHz)
				comb.	non-comb.	
idct	176	221	7.8	38	12	100
alpha blender	119	189	10.2	123	149	100
video filter	190	659	21.7	1,287	747	50
ATM se-engine	403	611	35.3	1,359	693	100
specific-purpose allocator	85	135	7.3	33	19	100
general-purpose allocator	297	353	9.9	204	80	100

Table 5.1: Result of the synthesis using target library tsmc.35u (area in library units).

CHAPTER 6. OPTIMIZATION OF LOADS AND STORES

In the previous two chapters, we have seen how pointers can be removed after pointer analysis. Now, we optimize the code to reduce the amount of storage necessary before loads (...=`*p`) and stores (`*p`=...). These optimizations are targeted at optimizing pointer variables pointing to multiple variables, which is quite common in the context of synthesizing functions, as shown in Appendix A.

In this section, the following assumptions are made in addition to the ones listed in Section 3.3. The pointer `p` is a scalar variable. Its points-to set consists of a set of variables (i.e. mapped to registers or wires in the physical implementation). The optimizations presented here are only performed when the previous assumptions hold. In Section 4.1, we have seen that location sets representing a single location can be mapped to variables. As a result, the optimizations presented here also apply to location sets representing a single location.

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 the optimizations presented here 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.

6.1 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 6.1. *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 6.1.

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 path [1,45].

Example 6.1. *In Figure 6.1, the load (*`out=*p`*) where* `p` *may point to* `a`*,* `b`*, or* `c`*, is replaced by an assignment from* `star_p`*. 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.*

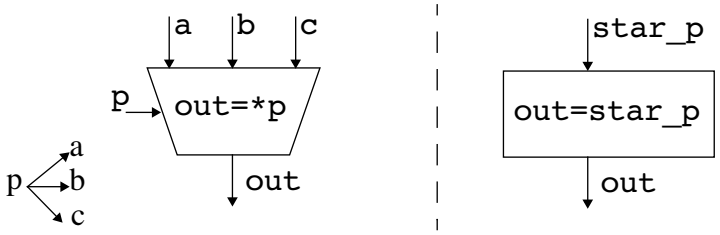


Figure 6.1: Optimization of a load

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 [1,45] 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 therefore is not necessary. So, 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. An outline of the complete algorithm for the optimization of loads follows:

1) Update `star_p` when `p`, or any variable of the points-to set changes.
2) Do live variable analysis [1,45] (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 6.2. *Let us take the code segment shown on Figure 6.2, before and after optimization, where the pointer* `p` *may point to* `a`, `b`, *or* `c`.

```
/* original code */ | /* code after optimization */
a=in;               | a=in;
wait();             | // if (p_tag==0) star_p=a;  // deadcode
temp=a+b+c;         | wait();
wait();             | temp=a+b+c;                 // a,b,c live
out=*p+temp;        | 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;
```

Figure 6.2: Example of code segment before and after optimizing load

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 removes 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 trade-off here between the number of live variables (i.e registers) and the amount of steering logic in the hardware implementation.

6.2 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 6.3. *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 5 variables {*`p`*,* `in`*,* `a`*,* `b`*,* `c`*} live before the store. Now assume that, at run time,* `p` *points to* `a`*. Since the value of* `a` *is going to be redefined by the store, it is not needed before the store. As a result the number of live variables before the store could be reduced by 1. Note that the same applies when* `p` *points to* `b` *or* `c`*.*

As we have seen in Example 6.3, 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 `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:

```
{ _star n _p, for n in {1, 2, ..., (pts_size-1)}}
```

("`_star`*n*`_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 `_star`*n*`_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 6.4.

Example 6.4. *If* `p` *may point to* `a`*,* `b`*, or* `c`*, we create* `_star1_p` *and* `_star2_p` *and define them as follows (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 5 live variables (cf.* Example 6.3) *can be replaced by the following code segment which uses only 4 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 6.1. Indeed, one could imagine an algorithm where the `_star`*n*`_p` variables are used at each store and defined when `p` or any variable of the points-to set is modified. Since each of the `_star`*n*`_p` variables 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 this thesis, a conservative approach is taken 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 6.5. *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) { // p==&a
  star_p = a;
  _star1_p = b; }
else {     // p==&b
  star_p = b;
  _star1_p = a; }

star_p = star_p + 1;

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

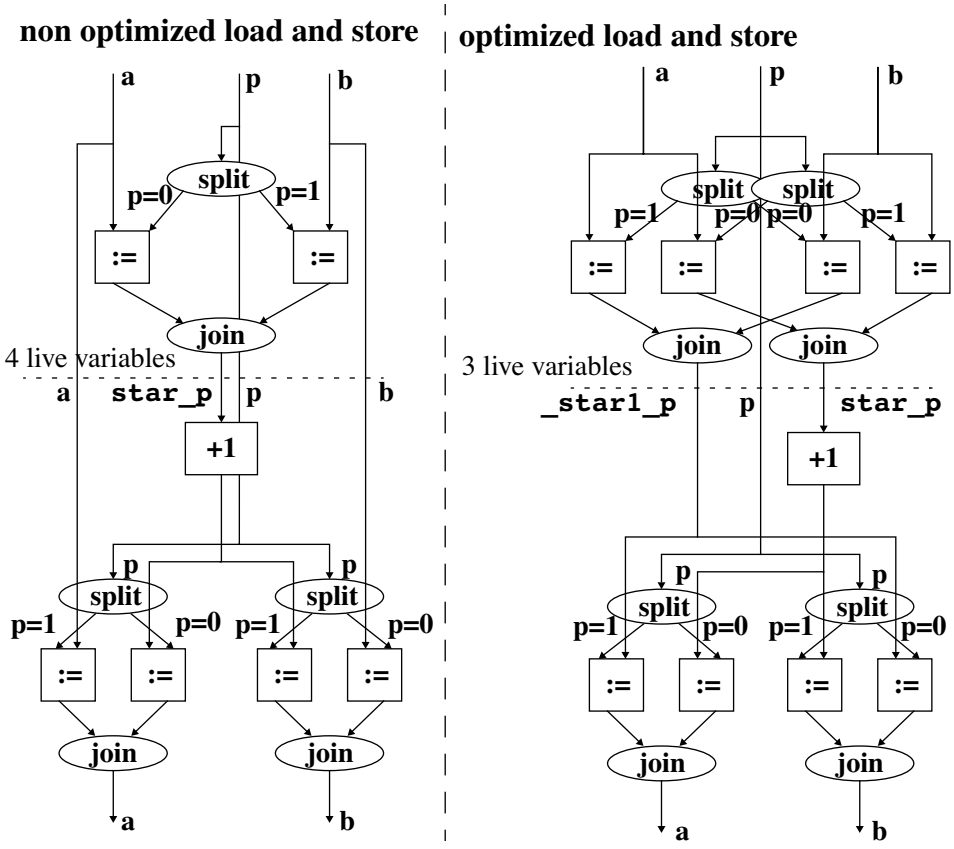


Figure 6.3: CDFG for `*p=*p+1` with `p->{a,b}`

Figure 6.3 shows the control/data-flow graph (CDFG) before and after optimization. The definitions of the temporary variables `star_p` *and* `_star1_p` *have 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 4* `{a,b,p,star_p}` *to 3* `{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[1] by loads from the same pointer (implemented as a forward data-flow analysis [1,45]).
2) List the loads post-dominated[2] by stores from the same pointer (implemented as a backward data-flow analysis [1,45]).
3) Do live-variable analysis assuming that each store in the list generated at Step 1 kills all variables in the points-to set.
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 `_star`*n*`_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 `_star`*n*`_p` variables to update the values of variables in the points-to set after the stores.

1. Instruction *d* dominates instruction *i* in a flow graph if every possible execution path from the *entry* node to *i* includes *d* [45].
2. Instructions *i* post-dominates instruction *d* in a flow graph if every possible execution path from the *d* to the *exit* (aka *sink*) node includes *i* [45].

Even though this optimization reduces the number of live variables before stores by at most one, it helps reducing the number of registers. There is however a trade-off between the number of registers used and the amount of steering logic. This optimization can be performed while optimizing the loads.

6.3 Results

We have written several models to study the effects of the different optimizations presented in this section.

This set of results illustrates the effects of each feature of the optimizer. Table 6.1 and Table 6.2 show the examples with the area and cumulative timing after pointer resolution with and without optimization.

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)

Table 6.1: Area after synthesis and optimization using target library lsi_10k (in library units).

example	**C lines**	**timing**	
		no optimization	**with optimizations**
`load`	43	46 ns	51 ns
`load/store`	48	86 ns	88 ns

Table 6.2: Timing after synthesis and optimization using target library lsi_10k (in ns).

The first model (`load`) tests the optimization of loads. It contains one pointer that may point to 3 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 is used. In the other path, all variables of the points-to set become live. Without any optimization we have five 32bit registers (i.e. 2334 units of non-combinational area). After optimization the number of registers is reduced to three (i.e. 1523 units of non-combinational area). This reduction of the storage goes however 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 trade-off between the number of registers and the amount 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.

6.4 Summary

In this chapter, compiler techniques were presented to reduce the number of live variables (akin to registers in the final design) in the code. Loads are optimized by creating a temporary variable that stores the value of the variable the pointer points to. In a way, this is similar to prefetching. Stores are optimized by creating temporary variables that stores the value of the variables the pointer does not point to. In Appendix A, we will see how synthesizing functions motivates such optimizations. Results show that, in general, there is a trade-off between storage (number of registers) and combinational logic. The optimizations reduce storage area with a possible increase in combinational logic area.

CHAPTER 7. ENCODING OF POINTERS

In software, the pointers' values represent addresses in memory space. 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 4.2.1, we have seen that the encoding of a pointer consists of two fields, 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 [4,47] could be used to encode the index part, but they are not addressed in this thesis.

Definition 7.1. *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 4.5, when pointers are assigned or compared, we may have to add `case` statements to "translate" the values of the pointers

by means of some combinational circuit. Encoding techniques can be used to minimize the size of these circuits. The goal is twofold: 1) 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; 2) 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 I call a *local encoding* in which two pointers that point to the same location set may have different encodings. Finding an exact solution is hard and thus a heuristic is 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.

7.1 Definition of the Problem

In this section, we present the problem of encoding the value of the pointers. The first goal is to minimize the size of the pointers. For two pointers `p` and `q`, when one pointer is assigned to the other (`p=q`) or when they are compared (`p==q`), the corresponding tags shall be equal (e.g., `p_tag==q_tag` when `p` and `q` reference the same location) or "as close as possible" to each others. If two tags have different bit widths, 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 7.2. *For two pointers, p_i and p_j, the* pointer dependence relation $r(p_i, p_j)$ *is 1 if and only if the two pointers are assigned or compared (otherwise it is 0).*

Definition 7.3. *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 if one pointer is assigned to the other of if they are compared.*

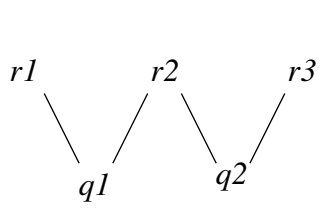
Example 7.1. *Consider the following code segment:*

```
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`.

This leads to the pointer-dependence graph on Figure 7.1a.



(a)

$r1 \rightarrow \{a,b\}$
$r2 \rightarrow \{b,c\}$
$r3 \rightarrow \{c,d\}$
$q1 \rightarrow \{a,b,c\}$
$q2 \rightarrow \{b,c,d\}$

(b)

Figure 7.1: (a) Example of pointer-dependence graph and (b) definitions of the points-to sets of each pointer

Note that the example presented here is equivalent to the more realistic Example A.6 in which functions mapped to components are shared.

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. The constraints on the encoding are two: 1) the supercube[1] of the codes of the symbols in each set must have minimum size; 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 7.2. *In Example 7.1, the pointers* `r1`, `r2`, *and* `r3` *may point to two different variables and* `q1`, `q2` *may point to three different variables. As a result we want to encode pointers* `r1`, `r2`, *and* `r3` *on 1 bit and pointers* `q1` *and* `q2` *on 2 bits.*

Figure 7.2a shows an example of a non-optimal 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 on Figure 7.2a.*

1.The *supercube* of a set of cubes is the smallest cube containing all the cubes in the set [14].

To find an optimal encoding, we look at the dependences between the pointers. Pointer `q1` *may take the value of* `r1` *or* `r2`*. So 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 Figure 7.2b.*

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

7.2 Problem Formulation

Let us consider P pointers $\mathrm{P}=\{p_1, p_2, \ldots, p_P\}$. For each pointer $p_i \in \mathrm{P}$, let Π_i be its points-to set. The points-to set Π_i 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

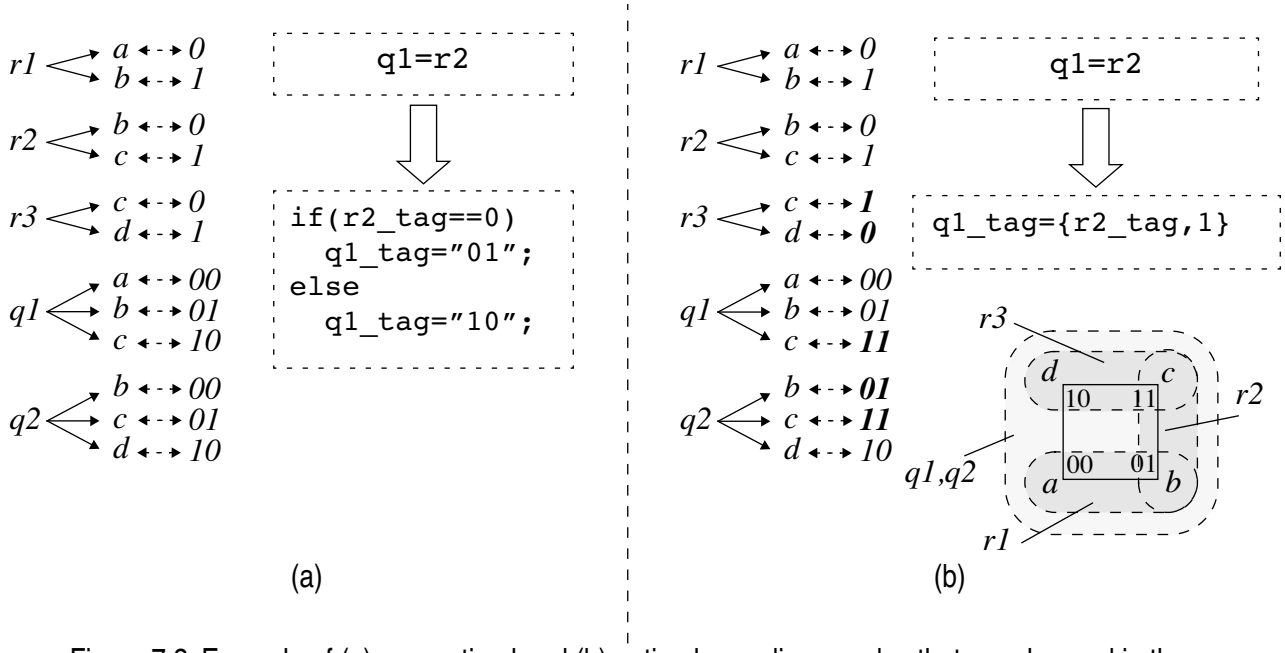


Figure 7.2: Example of (a) non-optimal and (b) optimal encodings; codes that are changed in the optimal encoding are shown in bold.

of the points-to set Π_i. The encoded values of the symbols in each set are noted $\{e^i_1, e^i_2, ..., e^i_{N_i}\}$.

Definition 7.4. *Two sets* Π_i *and* Π_j *are said to be* dependent *if their associated pointers are dependent (Definition 7.2).*

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 the supercubes is also minimized:

$$min\left(\sum_{i=1}^{P} dim(supercube(E_i))\right) \tag{7.1}$$

Example 7.3. *In the encoding presented in both Figures 7.2a and 7.2b,* $\sum_{i=1}^{P} dim(supercube(E_i)) = 1+1+1+2+2=7$ *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:

$$min\left(\sum_{i=1}^{P}\sum_{j=1}^{P} r(p_i, p_j) dist(E_i, E_j)\right) \tag{7.2}$$

where $dist()$ 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, $dist()$ is defined as:

$$dist(E_i, E_j) = \min_{perm()}\left(\sum_{k=1}^{N} H(perm(e^i_k), e^j_k)\right) \tag{7.3}$$

where $N=N_i=N_j$ is the number of symbols in the points-to sets, $perm()$ is in the set of the permutation functions of n bits, and $H(a, b)$ is the 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. The exact definition of the distance metrics in this case is presented in Appendix B. For example, the distance between two sets whose encodings have different lengths can be computed by padding the shorter code with 0s. Then, if the points-to sets differ, we are only interested in the distance between the encodings of the symbols common to the two points-to sets.

Our goal is to minimize Eq. 7.1 and Eq. 7.2. There is a trade-off 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 Eq. 7.2 and then Eq. 7.1. In general, we can cast the problem as follows:

$$min\left(\beta \sum_{i=1}^{P} dim(supercube(E_i)) + (1-\beta) \sum_{i=1}^{P} \sum_{j=1}^{P} r(p_i, p_j) dist(E_i, E_j)\right) \qquad (7.4)$$

where β is a coefficient between 0 and 1.

Since this problem is computationally hard to solve exactly, a heuristic solution is used.

7.3 Simplified problem

7.3.1 Formalism for a Global Solution

In the general formulation of the problem presented in Section 7.2, 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 encoded 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 7.4. *Figure 7.2a 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`.

Figure 7.2b gives an example of a better global encoding. The encoding is global because the pointers initially share the same encoding shown in Figure 7.3. 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 follows. Pointer `r2` *may point to* `b` *or* `c`. *In the global encoding, value* `01` *is*

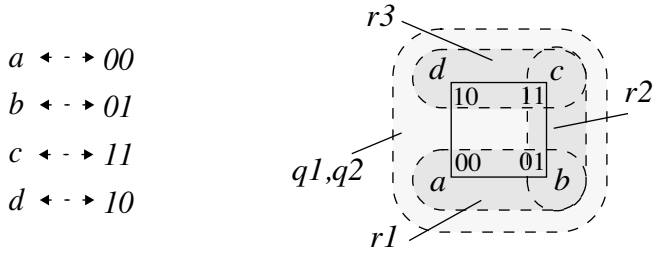


Figure 7.3: Global encoding and selection of the relevant bits for each pointer

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 reduced to 1 bit. Similarly, the size of r1 and r3 can be also be reduced to 1 bit.*

For a global encoding, minimizing Eq. 7.2 is 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. $dist(E_i, E_j) = 0 \ \forall (i, j) \in \{1, 2, ..., P\}^2$). The complexity of the logic to perform assignments and, to some extent, comparisons 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 Eq. 7.1 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*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 [13,62].

Definition 7.5. *The* relation matrix **A** *is defined as the matrix in which the rows represent the points-to sets and the columns the symbols. Entry* $a_{i,j}$ *of* **A** *is 1 if and only if the symbol* s_j *is in the set* Π_i.

Example 7.5. *Let's take the case of Example 7.1 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:*

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

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

We search for an encoding matrix **E**. Namely, each row in **A** corresponds to a points-to set. For each row α of **A**, we want the supercubes of the rows of **E** corresponding to the 1s in α to have minimum size. This corresponds to the constraint expressed in Eq. 7.1. This problem corresponds to the input encoding problem [13,14,62] if the 0s in matrix **A** are replaced by *don't care*s (i.e. *). In other words, our problem is a simpler instance of the general input encoding problem.

7.3.2 Global Encoding Algorithm

The problem of input encoding has been extensively studied [3,19,13,46,51,52,53,62]. We use an approach reminiscent of MUSTANG [46] and POW3 [3].

Definition 7.6. *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* **A**. *The weight* $w_{i,j}$ *on the edge* $\{s_i, s_j\}$ *is defined as:*

$$w_{i,j} = \sum_{k=1}^{P} a_{k,i} \cdot a_{k,j} \cdot (1 + \lceil Log_2 N \rceil - \lceil Log_2 N_k \rceil) \qquad (7.5)$$

where P *is the number of pointers,* N *is the total number of symbols,* N_k *the number of symbols in the set* Π_k, *and* $a_{i,j}$ *is an element of the relation matrix.*

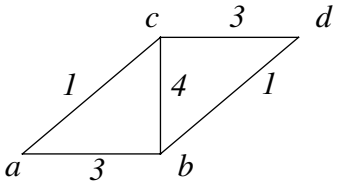
The weight $w_{i,j}$ in the affinity graph increases with the number of sets that contain both s_i and s_j: when two location sets are in many points-to sets, we want their codes to be close. This is even more important for small points-to sets. For example, if we have $N_k = 2$ symbols in the points-to 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 $\lceil log_2(N_k) \rceil = 4$. 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 $(1 + \lceil log_2 N \rceil - \lceil log_2 N_k \rceil)$.

The pointer encoding problem can be solved as an embedding of the affinity graph in the Boolean hypercube as done in [3,27,46,51,56].

Example 7.6. *The relation matrix presented in Example 7.5 (cf. Figure 7.4a) can be used to generate the affinity graph of Figure 7.4b.*

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

(a)



(b)

Figure 7.4: Example of (a) relation matrix and (b) corresponding affinity graph

Let's 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 Figure 7.5 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 4). The encodings of symbols* `a` *and* `b` *are also next to each other and so are the encodings of* `c` *and* `d`.

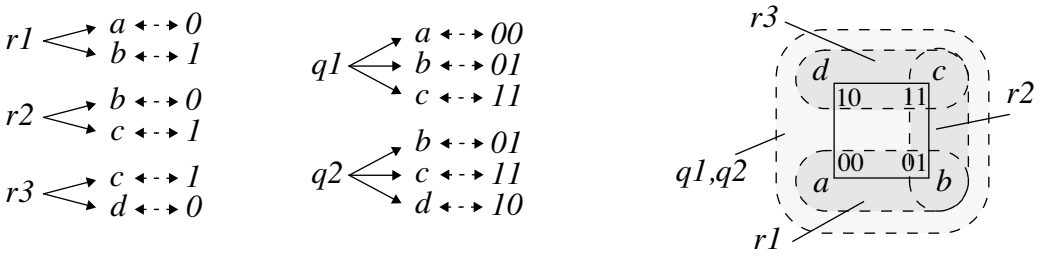


Figure 7.5: Example of optimal encoding

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 7.4 in a technique called *folding*. One symbol can also have multiple codes. The notion of *splitting,* presented in Section 7.5, is based on this property.

7.4 Encoding with folding

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

Definition 7.7. *We define as* folding *the action of assigning the same code to two different symbols.*

Proposition 7.1. *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 **A**, folding the symbols s_i and s_j is equivalent to replacing columns *i* and *j* by one column *k* such that:

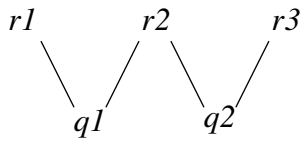
$$a_{k,l} = a_{i,l} \vee a_{j,l} \text{ for } l \text{ in } \{1, 2,..., N\}. \tag{7.6}$$

In the affinity graph, folding is done by merging (or fusing[1]) 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:

$$w_{k,l} = w_{i,l} + w_{j,l} \text{ for } l \text{ in } \{1, 2,..., N\}. \tag{7.7}$$

Graph-embedding techniques can be modified to incorporate folding. In Section 7.6, we present a column-based encoding algorithm with folding.

Example 7.7. *Let's consider the pointer-dependence graph on Figure 7.6, where* `r1`, `r2`, *and* `r3` *point respectively to {*`a`,`b`,`c`*}, {*`b`,`c`,`d`*} and {*`c`,`d`,`e`*}.*



r1 → *{a,b,c}*
r2 → *{b,c,d}*
r3 → *{c,d,e}*
q1 → *{a,b,c,d}*
q2 → *{b,c,d,e}*

Figure 7.6: Pointer-dependence graph and definitions of the points-to sets

1. 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 [16].

The relation matrix and the associated affinity graph are represented in Figure 7.7.

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

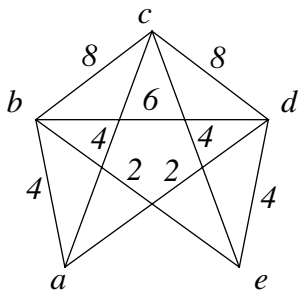


Figure 7.7: Relation matrix and corresponding affinity graph before folding

The number of variables (i.e. location sets) in each points-to set is either 3 (for `r1`, `r2`, *and* `r3`*) or 4 (for* `q1` *and* `q2`*). Therefore, we want to code the symbols associated with the variables on 2 bits. However, since we have 5 symbols, an encoding with less than 3 bits cannot be found without folding.*

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 on Figure 7.8.*

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

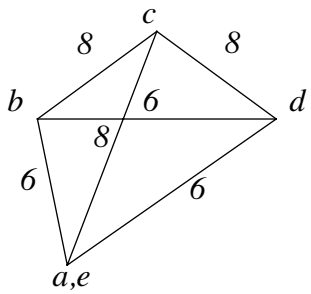


Figure 7.8: Relation matrix and corresponding affinity graph after folding `a` and `e`

This leads to an encoding that requires only two bits:

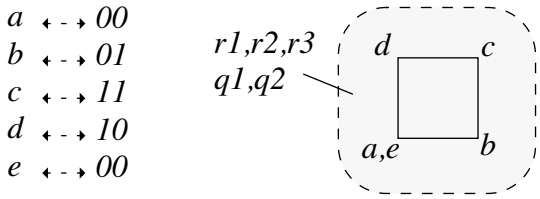


Figure 7.9: Result of the encoding after folding `a` and `e`

7.5 Encoding with splitting

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

Definition 7.8. *We define as splitting the action of assigning two or more codes to one symbol (or location set).*

In Section 7.3 and 7.4, 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 7.2. *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 7.8. *Let's consider the pointer-dependence graph on Figure 7.10 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 Figure 7.11.*

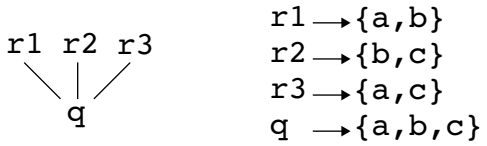


Figure 7.10: Pointer-dependence graph and definitions of the points-to sets

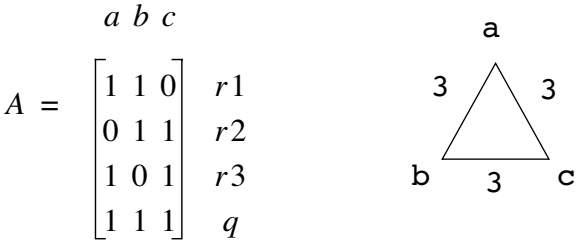


Figure 7.11: Relation matrix and corresponding affinity graph before splitting

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

Using the encoding technique without splitting symbols, we can find the encoding on Figure 7.12.

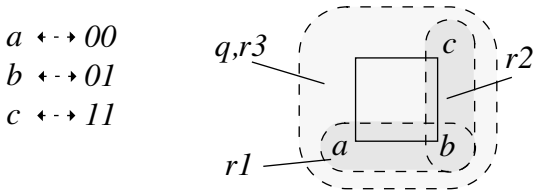


Figure 7.12: Result of the encoding without splitting

In this case `r1` *and* `r2` *are encoded on 1 bit but the encoding of* `r3` *requires 2 bits.*

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

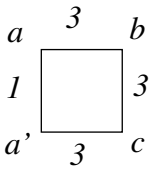


Figure 7.13: Relation matrix and corresponding affinity graph after splitting symbol a

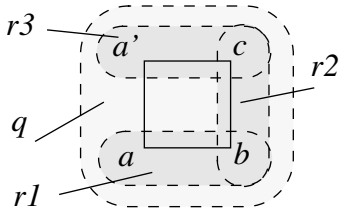


Figure 7.14: Result of the encoding after splitting symbol a

After splitting the symbol a*, we end up with the two symbols* a *and* a'*. The new encoding problem is presented on Figure 7.13. We can find the encoding on Figure 7.14 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 on Figure 7.14 is optimal: r1, r2, and r3 are encoded on 1 bit and the assignments to q *(*q=r1*,* q=r2*,* q=r3*) do not require any additional logic.*

As described in Section 7.2 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. Eq. 7.1), 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 7.2. 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:

$$E_k' = ((E_k - \{e_i\}) \cup E') \text{ where } E' \text{ is either } \{e_i\}, \{e_i'\} \text{ or } \{e_i, e_i'\}. \quad (7.8)$$

In order to minimize Eq. 7.1, for every set Π_k' that may contain s_i or s_i', we want to minimize

$$dim(supercube(E_k')) \quad (7.9)$$

which corresponds to

$$\min_{E'}(dim(supercube((E_k - \{e_i\}) \cup E'))) \text{ where } E' \text{ is either } \{e_i\}, \{e_i'\} \text{ or } \{e_i, e_i'\}. \quad (7.10)$$

In the relation matrix **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 $(a_k^i, a_k^{i'})$ is set to (0,1), (1,0), or (1,1) according to Eq. 7.10. If Eq. 7.10 achieves its minimum for the three values $\{e_i, e_i'\}$, $\{e_i\}$, and $\{e_i'\}$, then we select $\{e_i, e_i'\}$. Example 7.9 illustrates the reason of 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 7.6.

Example 7.9. *In Example 7.8, for the points-to set of* `r3`*, Eq. 7.10 is minimum for E'={*`a'`*}, the dimension of the supercube of the encoded symbols in the new points-to set is minimum equal to 1 when it contains* `a'` *only. As a result, in the relation matrix on Figure 7.13, the*

entry a_3^1 is set to 0 and a_3^2 is set to 1. For the points-to set of `q`*, Eq. 7.10 is minimum (equal to 2) 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_4^1 and a_4^2 are both set to 1. Since* `a` *is in the new points-to set of* `r1` *and* `a'` *in the new point-to set of* `r3`*, this allows to implement both* `q=r3` *and* `q=r1` *trivially.*

7.6 Encoding Algorithm

We propose a column-based approach such that the encoding matrix can be found column by column [13,14,18]. Our algorithm without folding and splitting is similar to the one used in POW3 [3]. The pseudo code of the algorithm with folding and splitting is presented on Figure 7.15.

The algorithm encodes the pointers with n bits 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_i^k for k={1, 2,..., n}. At each iteration k, we construct the k^{th} column of the encoding matrix **E** by assigning bit e_i^k to all symbols for i={1, 2,..., 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 7.9. *There is a* class violation *at iteration k when more than 2^{n-k} symbols have the same code so far.*

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.

```
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(∃s_j s.t. Proposition 7.1 verified and e_i==e_j){
         fold(s_i, s_j);  remove s_i;
         return true;
      }
      return false;
   }
```

Figure 7.15: Graph embedding algorithm with splitting and folding

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 end-points, starting with the edges with highest weights. The weights at each iteration are adjusted using the following formula [3]:

$$w_{i,j}^{new} = w_{i,j} \cdot (H(e_i, e_j) + 1) \tag{7.11}$$

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 7.9). 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 in Proposition 7.3 below.

Definition 7.10. *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 7.3. *An edge* $\{s_i, s_j\}$ *is violated at iteration k if either one of the following conditions applies:*

- *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 7.1 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 7.10. *Consider the problem presented on Figure 7.16. The associated relation matrix and affinity graph are presented on Figure 7.17 in which pointer* `q1` *may take the value of* `r1`, `r2`, *or* `r3` *and* `q2` *may take the value of* `r3`, `r4`, *or* `r5`.

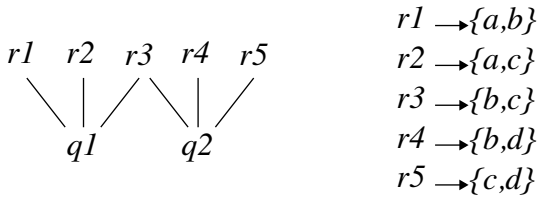


Figure 7.16: Pointer-dependence graph and definitions of the points-to sets

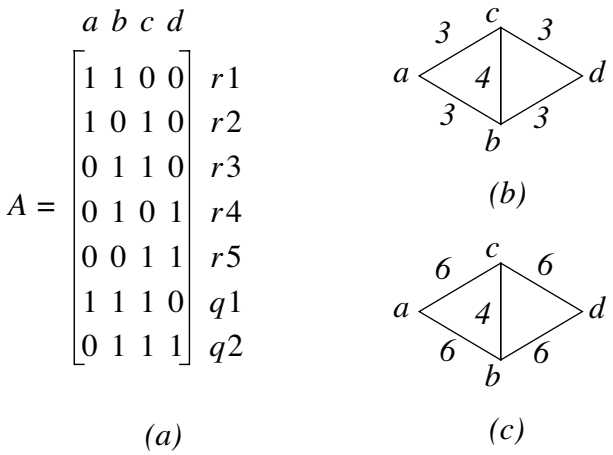


Figure 7.17: (a) Relation matrix and (b) affinity graph at the beginning of iteration 1; (c) affinity graph at the beginning of iteration 2

Since we have four symbols, we want to encode them on n=2 bits. The encoding is computed in two iterations. After the first iteration, at most two symbols may 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 two bits 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:*

$$A = \begin{array}{c} \begin{matrix} a & a' & b & c & d \end{matrix} \\ \begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 \end{bmatrix} \end{array} \begin{matrix} r1 \\ r2 \\ r3 \\ r4 \\ r5 \\ q1 \\ q2 \end{matrix} \tag{7.12}$$

We have 3 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*

$$A = \begin{array}{c} \begin{matrix} a & a' & b & c & d \end{matrix} \\ \begin{bmatrix} 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 \end{bmatrix} \begin{matrix} r1 \\ r2 \\ r3 \\ r4 \\ r5 \\ q1 \\ q2 \end{matrix} \end{array} \tag{7.13}$$

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:*

$$A = \begin{array}{c} \begin{matrix} & a' & \\ a & d & b & c \end{matrix} \\ \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{bmatrix} \begin{matrix} r1 \\ r2 \\ r3 \\ r4 \\ r5 \\ q1 \\ q2 \end{matrix} \end{array} \tag{7.14}$$

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:*

$$A = \begin{array}{c} \begin{matrix} d' & a' & & \end{matrix} \\ \begin{matrix} a & d & b & c \end{matrix} \\ \left[\begin{array}{cccc} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{array}\right] \end{array} \begin{array}{l} r1 \\ r2 \\ r3 \\ r4 \\ r5 \\ q1 \\ q2 \end{array} \tag{7.15}$$

We end up with the encoding on Figure 7.18 in which all constraints are satisfied.

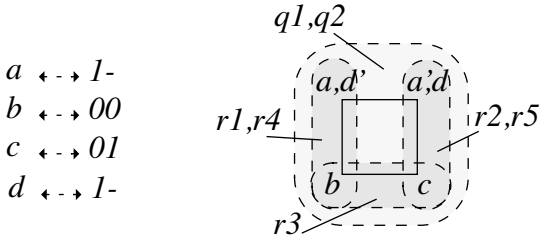


Figure 7.18: Result of the encoding after splitting and folding (where '-' is a *don't care*)

7.7 Results

The encoding algorithm presented here are compared with "traditional" encoding schemes for a set of examples. The results are presented in Table 7.1. 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 encoding schemes.

example (P/N)	encoding	storage	assign-ment	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	17,626
	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

Table 7.1: 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.

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 pointers' values. 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 this section, without splitting and folding. The bit-width of the pointer is then

shorter but can be further reduced. 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 non-optimal encoding used in Example 7.2), each variable in each points-to set is simply associated with an integer (e.g., 0 for the first variable, 1 for the second variable, etc...). The number of bits (bit-width) used to encode each tag is then minimum but the size of the circuit which translates the values of the pointers is not. Finally one-hot (*1-hot*) encoding gives larger codes, however the specific properties of the resulting codes can be used to simplify the decoding logic especially in loads and stores.

7.8 Summary

This chapter formulates the problem of encoding the values of the pointers. A solution based on graph embedding techniques is presented. The algorithm optimizes the storage area and the complexity of the decoding logic in loads and stores by reducing the bit-width of the pointers. At the same time the circuits implementing assignments and comparisons of pointers are also optimized by reducing the distance between dependent encodings. These optimizations enable the efficient synthesis of pointers especially for the synthesis of functions as it will be presented in Appendix A.

CHAPTER 8. LIBRARY OF ALLOCATORS AND OPTIMIZATION OF MALLOC AND FREE

In the Section 4.3 we have seen how `malloc` and `free` can be implemented using hardware memory allocators. Each allocator can perform both memory allocation and deallocation. A library of such allocators is provided. The designer may then to select the allocator architecture most suitable to the application. The library of allocator components contains three basic types of allocators. First a *general-purpose* allocator can allocate blocks of any size. The architecture of an *optimized general-purpose* allocator for which the deallocation scheme is optimized for latency is presented in Section 8.1. When the size of the block to be allocated is a fixed constant, the architecture of the allocator can be greatly simplified. The *specific-purpose* allocator presented in Section 8.2 can be used in such case.

The designer could also add new allocators in the library. The basic allocators presented here can be modified (e.g. to change the allocation or deallocation schemes, allocate a larger number of blocks or handle new sizes of elements) and added to the library.

Other types of allocators such as the ones described by Chang et al. [12] and Wuytack et al. [70] could also be added as new components in the library.

In some cases, the code can also be optimized. Calls to `malloc` and `free` can be removed and memory allocation can be done statically. In Section 8.3, a compiler technique is presented to automatically remove some of the dynamic memory allocation for sequences of `malloc` and `free`.

8.1 Optimized general purpose allocator

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

In order to simplify the process of looking up for a given block during deallocation, we propose to encode the characteristics of the allocated block inside of the pointer's *tag*. In

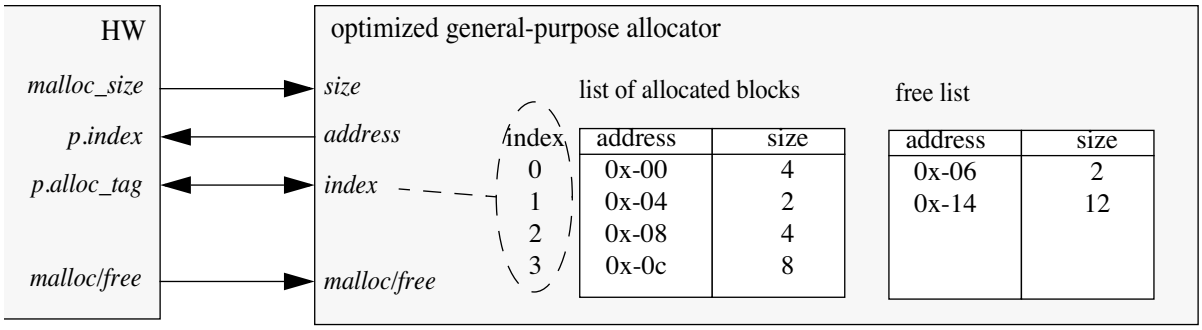


Figure 8.1: Architecture of an optimized general-purpose allocator

our implementation shown on Figure 8.1, the allocator stores the list of allocated blocks in an array. The index corresponding to the allocated block in this array is then encoded in the pointer's value. During deallocation, this index is sent to the allocator. The allocator can then directly find the allocated block according to this index, without having to search the entire array. The resulting optimized allocator is called *optimized general purpose*.

The encoded value of a pointer consists then of three fields: the *allocation tag*, the *tag* and the *index*. For a pointer `p`, the *tag* `p.tag` and the *index* `p.index` are defined as in Section 4.2. The *allocation tag* `p.alloc_tag` corresponds to the index of the block inside of the list of allocated blocks. In the implementation presented in this thesis, the allocation tag corresponds to the 8 most significant bits in the pointer's value, the tag corresponds to the following 8 bits and the index corresponds to the 16 least significant bits (as defined in Section 4.2.1). Figure 8.2 shows how the different field are laid out for an array of pointers.

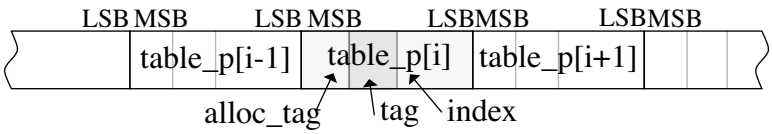


Figure 8.2: Encoding of pointers in an array for optimized general-purpose allocator

8.2 Specific purpose allocator

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

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

8.3 Optimizing sequences of `malloc` and `free` calls

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

Example 8.1. *Consider the following code segment.*

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

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

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

The optimization can be performed under two conditions. First, the size of the block to allocate has to be constant. If the size of the block to allocate is not known at compile-time, a *general purpose* or *optimized general-purpose* allocator would have to be used. Second, if a block is allocated within an unbounded loop, it has to be deallocated within the same unbounded loop. Using the results of pointer analysis, it is possible to implement a dataflow analysis [45] that finds at compile time the `malloc` and `free` calls that can be optimized (i.e. removed).

An brief outline of how the analysis is conducted follows. For each dynamically-allocated location set (i.e. each `malloc` call in the example), a counter is defined. The analysis steps thought the flow-graph of the procedure. The counter is incremented each time an element of the corresponding location set is allocated. Subsequently, each time an element of the location set is deallocated (result from pointer analysis), the associated counter is

decremented. Location sets allocated and not deallocated within the same loop can be found. The `malloc` and `free` corresponding to these locations cannot be optimized. Otherwise, they can be optimized.

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

8.4 Experimental results and discussion

For the set of examples presented here, three types of allocators have been synthesized as port of the library. In the results presented in Table 8.1, allocators are designed to allocate up to 16 blocks of memory. They are synthesized directly from C using SpC and Synopsys Behavioral Compiler [93]. The general-purpose allocators use *first fit* to allocate blocks and merge adjacent free blocks during deallocation. The first row presents the results for the *general-purpose allocator* without any optimization. The second row of Table 8.1 shows the size of the *optimized general-purpose allocator* for which the deallocation scheme has been optimized using the modified *tag* as presented in Section 8.1. Even though the complexity of the controller is reduced (from 52 states to 46), the size of the optimized allocator is roughly the same because of an increase in the steering logic. The latency of the deallocation task is however reduced as shown later in Table 8.2.

Finally the third row presents the results for the *specific-purpose allocator* introduced in Section 8.2. As expected its size is much smaller than the *general-purpose allocator*s.

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

Table 8.1: Implementation of the different allocators (area in library units using the tsmc.35 target library; *comb.* and *non-comb.* represent respectively the area of combinational logic and non-combinational logic (i.e. registers, etc.) at 100MHz)

Table 8.2 shows the results for four different examples. The first two examples *test1* and *test2* consists of three `malloc` calls and two `free` calls. All `malloc` calls allocate objects of the same constant size. Hence a *specific-purpose allocator* can be used. For the first example, all calls to `malloc` and `free` can be removed during optimizations. For the second example, one of the `malloc`s is called inside of an unbounded loop and cannot be removed. The third example is a filter used in the JPEG library of Synopsys COSSAP [93] and is used, for example, for RGB to YCrCb transformations. The filter implements the operation $\boldsymbol{Y}[i] = clip(\boldsymbol{A} \cdot \boldsymbol{X}[i] + B, C)$ for $i = \{1, 2, ..., n\}$, where $\boldsymbol{A}$ is a 3×3 matrix, B and C are vectors and $\boldsymbol{Y}$ and $\boldsymbol{X}$ are two $3 \times n$ dynamically-allocated matrices. Finally the last example is the implementation of an ATM segmentation engine. The segmentation engine receives frames to be sent from the host. These frames are segmented into 48 byte cells (payload of an ATM cell) to be transmitted on the network. The engine keeps track of each frame in a queue. For every new frame, a new virtual connection is opened and a new queue element is allocated. As a results, we have two sets of `malloc` calls: one to allocate queue elements, the other to allocate connection status records.

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

Table 8.2: Results for the different examples and optimizations (size in library units using the tsmc.35 target library; frequency 100MHz for test1, test2 and ATM, 50MHz for JPEG; CPU time for synthesis measured on Sun Ultra2 does not include high-level synthesis)

For each example, the first set of results illustrates the case where `malloc` calls are mapped to two *general-purpose* allocators (*no sharing*). For the ATM segmentation engine, two *specific-purpose* allocators are used instead of the *general-purpose* allocators. In the other results, one allocator is shared. As expected, the latency (measured by simulation at the RTL level) increases without sharing with a decrease in area. In Table 8.2, we

can also verify that the total latency of the design decreases when the *optimized general-purpose allocator* (*gen. alloc. optimized*) is used. The use of a *specific-purpose allocator* (*spec. alloc.*) when possible provides significant reduction both in latency and area. Finally, further optimizations can be performed when sequences of `malloc` and `free` calls can be removed (*sequence*).

8.5 Summary

In this chapter, different architectures of allocators were presented. These different memory allocators are implemented in a library of components. The different architectures are selected by the designer before synthesis. This fits into the application-specific memory management methodology outlined in Section 2.3. In addition to a general-purpose memory allocator, two other optimized allocators were presented. First the specific-purpose allocator, which may only allocate blocks of a fixed-constant size, is the simplest implementation of an allocator hence it is faster and smaller. Otherwise, in the optimized general-purpose allocator, information about the allocated blocks is encoded inside of the pointers' value to speed up deallocation. Finally simple compiler analyses are presented to replace sequences of `malloc` calls by static memory allocation.

CHAPTER 9. CONCLUSION

This thesis presented a methodology for efficiently mapping C code onto hardware. The tool SpC is the first solution for synthesizing C code with pointers, complex data structures and dynamic memory allocation/deallocation. It fits into current memory management methodology, enabling hardware implementations in which pointers may not only represent memory addresses but may also reference data physically mapped to registers, wires, ports, etc. For dynamic memory allocation, an application-specific library of hardware allocators was designed.

Several optimization techniques are also presented. Compiler optimizations are used to reduce the storage before loads and stores and eliminate dynamic memory allocations in sequences of `malloc` and `free`. An encoding algorithm, reminiscent of those used for FSM encoding in logic synthesis, was developed to efficiently encode the values of the pointers. Finally, a novel architecture of allocators was introduced in which the dealloca-

tion process is sped up by encoding some of the information of the memory block referenced inside of the pointer's value.

Future work is needed to better integrate SpC and existing tools both at the system level and architectural level. Based on the results of this research, one could define a complete C-based automated refinement methodology in which software models written in C could be directly synthesized. As an example, the synthesis of pointers would enable the synthesis of C functions with parameters passed by reference using HLS. This, however, would require a tighter integration between the tool resulting of this research, SpC, and HLS. More work is also needed to enable HLS methodology and algorithms to scale to thousands of lines of code involving nested function calls. The integration of SpC with existing system-level tools [74,80] would also greatly help software/hardware migration.

On the compiler side, several optimizations could be performed. Many designers and programmers make use of a specific coding styles (i.e. no out of bound array accesses, use of predefined data-structures [80], smart-pointers, etc.). Recognizing these coding styles would improve the accuracy of code analyses, which, in our case, may lead to improved memory representations. Pointer analysis could also be improved, for example, by looking more precisely at predicates to avoid false paths in the control flow. Moreover, compiler front ends and intermediate representation should be modified to better support C/C++ libraries for modeling hardware/software systems [75,81,86, 89].

Another possible extension of this work is on supporting different target architectures, both software (e.g. DSP, VLIW) and hardware (e.g. FPGA and other reconfigurable architectures). FPGA for example could be a good target architecture. The more restrictive design space on FPGAs, as compared to semi-custom ASICs, simplifies the task of HLS.

Moreover, designers who use FPGAs instead of ASICs are usually willing to trade off performance over time-to-market. Synthesizing FPGA logic directly from C would further reduce design time making hardware synthesis even more like software compilation.

Finally this research can be extended to other language constructs. The synthesis of pointers enables the synthesis of functions with parameter passed by reference (cf. Appendix A). It could enable the synthesis of object-oriented features found in C++ or Java [33].

REFERENCES

[1] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, "*Compilers, Principles, Techniques and Tools,*" Addison-Wesley Publishing Company, Reading, MA, 1986.

[2] Jonathan Babb, Martin Rinard, Andras Moritz, Walter Lee, Matthew Frank, Rajeev Barua, and Saman Amarasinghe, *"Parallelizing Applications Into Silicon,"* proc. IEEE Workshop on FPGAs for Custom Computing Machines '99 (FCCM '99), Napa Valley, April 1999.

[3] Luca Benini and Giovanni De Micheli, "*State assignment for low power dissipation*," IEEE Journal of Solid State Circuits, vol. 11, No. 4, pp. 258-268, March 1995.

[4] Luca Benini, Giovanni De Micheli, Enrico Macii, Donatella Sciuto and Christiana Silvano, "*Address Bus Encoding Techniques for System-Level Power Optimization,*" proc. Design Automation and Test is Europe DATE'98, pp. 861-866, Paris, France, February 1998.

[5] Ivo Bolsens, Hugo J. De Man, Bill Lin, Karl Van Rompaey, Steven Vercauteren, Diederik Verkest, "*Hardware/Software Co-Design of Digital Telecommunication Systems,*" Proceedings of the IEEE, Vol 85, No. 3, pp.391-418, March 97.

[6] C.T.Bye, M.R. Lightner and D.L. Ravenscroft, "*A Functional Modeling and Simulation Environment based on ESIM and C*," proc. ICCAD'84, pp.51-53, November 84.

[7] Timothy J. Callahan and John Wawrzynek, "*Instruction Level Parallelism for Reconfigurable Computing,*" proc. FPL'98, Tallinn, September 1998.

[8] Raul Camposano, W. Wolf, "*High-Level VLSI Synthesis,*" Kluwer Academic Publishers, Boston, 1991.

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

[10] C. Chu, M. Potkonjak, M. Thaler, and J. Rabaey. "*HYPER: An interactive synthesis environment for high performance real time applications*." In Proc. International Conference on Computer Design, pages 432-435, CA, October 1989. IEEE Computer Society Press.

[11] L. Claesen, F. Catthoor, D. Lanneer, G. Goossens, S. Note, J. van Meerbergen, and H. de Man. "*Automatic synthesis of signal processing benchmarks using the CATHEDRAL silicon compilers*." In Proc. IEEE Custom Integrated Circuits Conference, 1988

[12] J. Morris Chang, Edward R. Gehringer "A High-Performance Memory Allocator for Object-Oriented Systems," IEEE Trans. on Computers, vol. 45, no. 3, March 96.

[13] Giovanni De Micheli *"Symbolic Design of Combinational and Sequential Logic Circuits Implemented by Two-Level Logic Macros."* IEEE Transaction on CAD, volume 5(4), pp. 597-616, 1986.

[14] Giovanni De Micheli *"Synthesis and Optimization of Digital Circuits"*, Mc Graw Hill, Hightstown, NJ, 1994.

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

[16] Narsingh Deo, *"Graph Theory with applications to Engineering and Computer Science"*, Prentice-Hall, Englewood Cliffs, NJ, 1974.

[17] Alain Deutsh, "*Interprocedural may-alias analysis for pointers: Beyond k-limiting*," Proc. of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation, pp 230-241, June 94.

[18] T.A Dolotta and E.J. McCluskey, *"The Encoding of Internal States of Sequential Machines."* IEEE Transactions on Electronic Computers, volume EC-13, pp. 549-562, October 1964.

[19] C. Duff, *"Codage d'automates et théorie des cubes intersectants*," Thèse, Institut National Polytechnique de Grenoble, France, March 1991.

[20] R. Ernst, J. Henkel, Th. Benner, W. Ye, U. Holtmann, D. Herrmann, and M. Trawny, *"The COSYMA Environment for Hardware/Software Cosynthesis of Small Embedded Systems"* Microprocessors and Microsystems 20(3),pp.159-166, May 1996.

[21] David Evans, "*Static Detection of Dynamic Memory Errors*," Proc. SIGPLAN Conference on Programming Language Design and Implementation (PLDI '96), Philadelphia, PA, May 1996.

[22] Daniel Gajski, Frank Vahid, Sanjiv Narayan, Jie Gong, "*Specification and Design of Embedded Systems,*" PTR Prentice Hall, 1994

[23] Daniel Gajski, Nikil Dutt, Allen Wu and Steve Lin, "*High-Level Synthesis, Introduction to Chip and System Design,*" Kluwer Academic Publishers 1992.

[24] Rakesh Ghiya and Laurie Hendren, "I*s it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C*," Proc. 23th Annual ACM Symposium on Principle of Programming Languages.

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

[26] M. B. Gokhale and R. Minnich, "*FPGA Computing in a Data Parallel C,*"proc. IEEE Workshop on FPGAs for Custom Computing Machines (FCCM'93), Napa, CA, 1993.

[27] Evgueii Goldberg, Tiziano Villa, Robert Brayton, Alberto Sangiovanni-Vincentelli, *"Theory and Algorithms for Face Hypercube Embedding"* Transaction on CAD, volume 17(6), June 1998.

[28] Bjarne Hald, Jan Madsen, Ahmed Jerraya, "*A New Approach to Optimization and Reuse of Hierarchical Architectures*," submitted to IEEE Transaction on VLSI, 1998

[29] Ahmed Jerraya, Hong Ding, Polen Kission, Maher Rahmouni, "*Behavioral Synthesis and Component Reuse with VHDL,*" Kluwer Academic Publishers, Norwell, MA, 1997.

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

[31] H. Keding, M. Willems, M. Coors, H. Meyr, "*FRIDGE: A Fixed-Point Design And Simulation Environment*," proceedings of the Design Automation and Test in Europe DATE'98, pp. 429-435, 1998.

[32] Brian Kernighan, Dennis Ritchie, "*The C Programming Language*", Prentice Hall Software Series, Englewood Cliffs, NJ, 1988.

[33] Tommy Kuhn, Wolfgang Rosenstiel, and Udo Kebschull, "*Description and Simulation of Hardware/Software Systems with Java*," 36th Design Automation Conference (DAC), New Orleans, USA, 1999.

[34] Hwayong Kim, Kiyoung Choi, *"Transformation from C to Synthesizable VHDL*" proc. Asia Pacific Conf. on HDL APCHDL'98, July 1998.

[35] David Knapp, *"Behavioral Synthesis: Digital System Design Using the Synopsys Design Compiler"*, Prentice Hall, Upper Saddle River, NJ, 1996.

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

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

[38] Yanbing Li, Tim Callahan, Ervan Darnell, Randolph Harr, Uday Kurkure, Jon Stockwood, "*Hardware-Software Co-Design of Embedded Reconfigurable Architectures,*" proc. Design Automation Conference DAC'00, pp. 507-512, Los Angeles, June 2000.

[39] Stan Liao, "*Towards a new standard for system level design,*" proc. CODES'00, pp. 2-6, 2000.

[40] Stan Liao, Steve Tjang, Rajesh Gupta, "*An Efficient Implementation of Reactivity for Modeling Hardware in the Scenic Design Environment*," Proc. Design Automation Conference DAC'97, pp.70-75, June 97.

[41] S.-W Liao, A. Diwan, R. P. Bosch, Jr. and A. Ghuloum, M. S. Lam, *"SUIF Explorer: An Interactive and Interprocedural Parallelizer,"* Proceedings of the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'99), May, 1999.

[42] Elliot Linzer, Ephraim Reig, *"New Scaled DCT Algorithms for Fused Multiply/Add Architectures"*, proc. International Conference on Acoustics, Speech, and Signal Processing, Proceedings ICASSP '91, Vols.1-5, pp.2201-2204, 1991.

[43] O. Mencer, M. Morf, J. Flynn, "*PAM-Blox, High Performance FPGA Design for Adaptive Computing,*" proc. IEEE Symposium on FPGAs for Custom Computing Machines, Napa Valley 1998. http://umunhum.stanford.edu/PAM-Blox

[44] Philip Moorby, Donald Thomas, "The Verilog Hardware Description Language," Kluwer Academic Pub., 4th edition (May 1998).

[45] Steven Muchnick, "*Advanced Compiler Design and Implementation,*" Morgan Kaufman Publishers, San Francisco, California, 1997.

[46] A. R. Newton, S. Devaras, H-keung Ma and A. Sangiovanni-Vincentelli "*MUSTANG: State Assignment of Finite State Machines Targeting Multilevel Logic Implementations*." IEEE Transaction on CAD, volume 7(12), pp. 1290-1300, 1988.

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

[48] Douglas Perry, "*VHDL*", McGraw-Hill Companies, Inc.

[49] P. J. Plauger, "*The Standard C library,*" Prentice Hall Software Series, Englewood Cliffs, NJ, 1991.

[50] R. Rugina and M. Rinard., "*Pointer Analysis for Multithreaded Programs,*" proc. SIGPLAN '99 Conference on Program Language Design and Implementation, Atlanta, May 1999.

[51] G. Saucier, "*State Assignment of Asynchronous Sequential Machines using Graph Techniques*," IEEE Transaction on Computer, March 1972.

[52] G. Saucier, C. Duff, F. Poirot, "*State assignment using a new embedding method based on intersecting cube theory*," proc. Design Automation Conf., pp. 321-326, June 1989.

[53] G. Saucier, M.C. Depaulet and P. Sicard, "*ASYL: A rule-based system for controller synthesis*," IEEE Transactions on CAD, volume CAD-6, pp. 1088-1097, November 1987.

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

[55] Luc Séméria, Giovanni De Micheli, *"SpC: Synthesis of Pointers in C. Application of Pointer Analysis to the Behavioral Synthesis from C",* proc. IEEE/ACM International Conference on Computer Aided Design, ICCAD'98, pp. 340-346, November 1998.

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

[57] Luc Séméria, Koichi Sato, Giovanni De Micheli, "*Resolution of Dynamic Memory Allocation and Pointers for the Behavioral Synthesis from C*," Proc. Design Automation and Test in Europe DATE'00, pp. 312-319, Paris, March 2000.

[58] Luc Séméria, Abhijit Ghosh, "*Methodology for Hardware/Software Co-verification in C/C++*," Proc. Asia South Pacific Design Automation Conference ASP-DAC'00, pp. 405-408, Yokohama, January 00.

[59] Luc Séméria, Koichi Sato, Giovanni De Micheli, "*Memory Representation and Hardware Synthesis of C Code with Pointers and Complex Data Structures*," proc. Synthesis And System Integration of MIxed Technologies Workshop, SASIMI'00, pp. 43-48, Kyoto, April 2000.

[60] Bjarne Steensgaard "*Points-to Analysis by Type Inference of Programs with Structures and Unions*," proc. International Conference on Compiler Construction ICCC'96, pp.136-150, April 1996.

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

[62] Tiziano Villa, Alberto Sangiovanni-Vincentelly, "*Nova: State Assignment of Finite State Machines for Optimal Two-Level Logic Implementation,*" IEEE trans. on Computer-Aided Design, Vol. 9, pp.905-924, September 1990.

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

[64] Kazutoshi Wakabayashi, Takumi Okamoto, "*C-Based SoC Design Flow and EDA Tools: An ASIC and System Vendor Perspective*," IEEE trans. on Computer-Aided Design, vol 19, number 12, pp. 1507-1522, December 2000.

[65] Paul Wilson, Mark Johnstone, David Boles, "*Dynamic Storage Allocation: A Survey and Critical Review*," Int. Workshop Memory Management, Kinross, Scotland, Sept. 95.

[66] Robert Wilson, *"Efficient, Context-Sensitive Pointer Analysis For C Programs"*, PhD Dissertation, Stanford University, 1997.

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

[68] R.P.Wilson et al. "*Suif: An Infrastructure for Research on Parallelizing and Optimizing Compilers*", ACM SIPLAN Notices 28(9), pp.67-70, Sept. 1994.

[69] Sven Wuytack, Francky Catthoor, Hugo De Man, "*Transforming set data types to power optimal data stuctures*," IEEE Tansactions on Computer Aided Design, pp. 619-629, June 1996.

[70] Sven Wuytack, Julio L. da Silva Jr., Francky Catthoor, Gjalt de Jong, Chantal Ykman, "*Memory Management for Embedded Network Applications*," IEEE Transactions on Computer-Aided Design, Vol. 18-5, pp. 533-544, May 1999.

[71] Accelera C/C++ Class Library Standardization Working Group http://www.eda.org/alc-cwg/

[72] Arexys, http://www.arexys.com

[73] C Level Design, http://www.cleveldesign.com

[74] CoWare N2C, http://www.coware.com/n2c.html

[75] CynApps, http://www.cynapps.com

[76] EtherDesign Software, http://www.etherdesign.com

[77] Frontier Design A|RT BUILDER, http://www.frontierd.com/artbuilder.htm

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

[79] IEEE Computer Society Design Automation Technical Committee (DATC) http://www.computer.org/tab/datc/

[80] IMEC, Matisse, http://www.imec.be/vsdm/projects/matisse/

[81] IMEC OCAPI http://www.imec.be/ocapi/

[82] International Technology Roadmap for Semiconductors, http://public.itrs.net

[83] Get2Chip, http://www.get2chip.com

[84] LCLint, http://lclint.cs.virginia.edu

[85] Mentor Graphics Monet, http://www.mentor.com/monet/

[86] Open SystemC Initiative, http://www.systemc.org

[87] Rational Software Purify, http://www.rational.com

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

[89] SpecC Technology Open Consortium http://www.specc.gr.jp

[90] Suif compiler framework http://suif.stanford.edu/

[91] Synopsys Inc. Behavioral Compiler http://www.synopsys.com/products/bc/

[92] Synopsys Inc. CoCentric SystemC Compiler http://www.synopsys.com/products/cocentric_systemC/

[93] Synopsys Inc. Cossap http://www.synopsys.com/products/cossap/

[94] Transmogrifier http://www.eecg.toronto.edu/EECG/RESEARCH/tmcc/tmcc/

APPENDIX A. FUNCTIONS AND PASSING PARAMETERS BY REFERENCE

Functions are one of the fundamental constructs of programming languages. They add structure to a program and make it more readable by hiding some of the implementation details. They also enable reuse. The functionality described inside a function can be used at different points in a program. These arguments also apply to hardware behavioral descriptions in which functions are used to hide functionality (e.g. hide communication protocols) or reuse components (e.g. defining new complex operations). In this section, we look into ways of synthesizing C functions in hardware and study their interactions with the synthesis of pointers. The techniques presented here also apply to any language in which parameters are passed by reference, which include most programming languages (e.g. C++, Java, ADA, etc.).

A.1 Synthesis of Functions

In high-level synthesis tools, function are either inlined or mapped to components. In the case where they are inlined, each operation inside the functions is scheduled and mapped to a basic operator (e.g. multiplier, ALU, adder, etc.). These resources can then be

shared to implement multiple operations inside of a single thread of control (i.e. same process in VHDL, or same *always* block in Verilog). The advantages of such an approach are the following: 1) functions are straightforward to support; and 2) sharing can be performed at the operation level (fine grain), which may lead to more efficient scheduling and resource allocation. The disadvantages are: 1) a loss of structural information; and 2) an increased complexity of the scheduling and resource allocation problems (the size of the CDFG increased) for which algorithms may have an hard time finding a optimal solution. The COSMOS tool [29] and its commercial version from Arexis [72] use functions to hide the communication in a behavioral model. The idea follows the good practice of separating functionality (usually untimed before scheduling) and communications (which may be described in a cycle-accurate manner or not). Such function implementing communication protocols can indeed be inlined and considered as a new level of hierarchy when building the control state machine. However, when the functions inlined also describe functionality and not just communication, some information about the structure of the hardware to generate may be lost.

In a tool like Behavioral Compiler [91], functions can be used to explicitly define new operators implementing complex arithmetic functions such as a multiply accumulate (MAC) unit. In such case, instead of inline functions and inferring a separate multiplier and adder for a MAC, the designer may directly instantiate a MAC. Resource allocation and scheduling is then performed on the MAC itself instead of on separate multiply/add components. This eases the scheduling and allocation tasks. In such a case, functions are said to be mapped to components (or operators). Such components are then activated when

the function is called. The same component can be used at different points in the program, in such a case, we say that the component is *shared* by the different call sites.

Commercial synthesis tools usually have limitation on functions that can be mapped to components. Behavioral Compiler, for example, may only map combinational-logic functions. As a result, a function (or a task in Verilog HDL) may not contain infinite loops, *wait* statements or even calls to other functions mapped to components. This works fine for a simple component such as a MAC but would not scale to complex Intellectual Property (IP) blocks such as an IDCT or a PCI interface.

Some researches have been working on more elaborate ways of synthesizing functions. In the work of Hald et al. [28], functions may be implemented as multi-cycle operations and may even call other functions. However, only functions that are called within the same scope may be shared. Modules can then be instantiated in a hierarchical manner. One function is mapped to a module and subsequent functions called are mapped to operators instantiated within this module. To generalize this work, one can imagine a framework where functions called within different contexts may be mapped to a shared component.

Functions in C differ from their counterparts in Verilog or VHDL. In HDLs, functions are defined with a set of ports (`in`, `out` and `inout`) to which parameters are passed by value (i.e., a copy of each parameter is stored within the function). In C, pointers are used in function calls to pass parameter by reference. In this case, functions do not keep a copy of the value of parameters passed by reference internally. Example A.2 illustrates the difference between parameter pass by value and parameter pass by reference.

Example A.1. *We consider the following function definition in HDL:*

```
2_incr(inout a; inout b) {
    a = a + 1;
    b = b + 1;
  }
```

Now assume function `2_incr` *is called in the following context.*

```
    c = 0;
    2_incr(c,c);
```

In Verilog the parameter `c` *is passed by value. When the function is called, the value of* `c` *(namely 0) is assigned to the local variables* `a` *and* `b` *inside of the function* `2_incr`*. The value of* `a` *and* `b` *after executing the function are both equal to 1. As a result, the value of* `c` *after returning from* `2_incr` *is 1.*

Let us consider now consider the following C code in which variable `c` *is passed by reference:*

```
  2_incr(int *p; int *q) {
    *p = *p + 1;
    *q = *q + 1;
  }
```

where `2_incr` *is called in the following context:*

```
    c = 0;
    2_incr(&c,&c);
```

In this case the value of `c` *is not copied. The first line* `*p=*p+1` *increments the value of c from 0 to 1 and the second line* `*q=*q+1` *increments the value of* `c` *from 1 to 2. As a result, when* `c` *is passed by reference, the new value of* `c` *is 2.*

Synthesizing functions with parameters passed by reference required the synthesis of the underlying pointers. We have seen in Example A.1 that aliases must be disambiguated within the function called. Examples A.2 and A.3 show how pointers may be resolved within a function. They also show that the sharing information is also used to properly map functions to components. Note that, in general, pointers cannot be replaced by simple

in or inout ports. Synthesizing functions with parameter passed by reference through the use of pointers is not straightforward. Accurate pointer analysis is necessary to compute the set of locations the pointer may reference (points-to set) inside of the function. Information about resource sharing is also necessary.

Example A.2. *Assume the function* `2_incr` *in Example A.1 written in C, is used as follows.*

```
2_incr(int *p; int *q) {
  *p = *p + 1;
  *q = *q + 1;
}

main () {
  ...
  2_incr(&a,&b);
  2_incr(&c,&c);
  ...
}
```

If the function `2_incr` *is mapped to a component and only one of such component is instantiated (i.e. the* `2_incr` *operator is shared at the two call sites). One possible behavioral description in an HDL like language is the following. The corresponding data-flow graph is presented on Figure A.1.*

```
2_incr( in p_tag;
        in q_tag;
        inout main_a;
        inout main_b;
        inout main_c; )
{
  if(p_tag == 0)
    main_a = main_a + 1;
  else
    main_c = main_c + 1;

  if(q_tag == 0)
    main_b = main_b + 1;
  else
    main_c = main_c + 1;
}
```

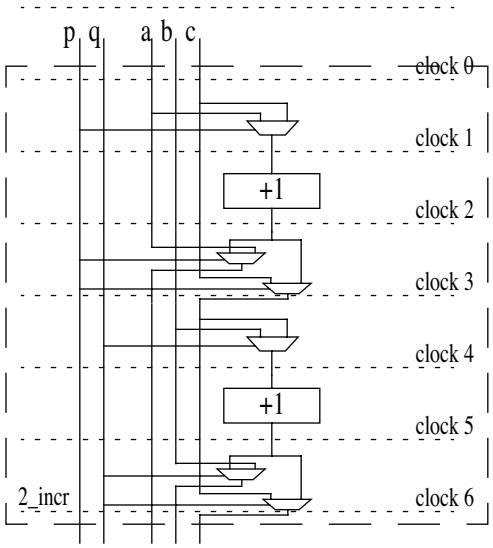


Figure A.1: Implementation of function `2_incr` with sharing

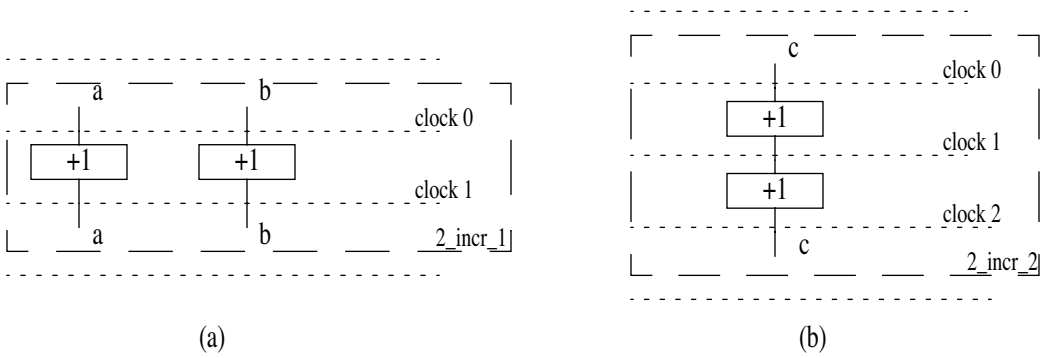


Figure A.2: Implementations of (a) function `2_incr_1` and (b) function `2_incr_2`

```
main () {
   ...
   2_incr(0,0,a,b,0);
   2_incr(1,1,0,0,c);
   ...
}
```

Now if the `2_incr` *operator is not shared, two components are instantiated. One possible behavioral description in an HDL-like language is the following. The data-flow graph for the two functions 2_incr_1 and 2_incr_2 is presented on Figure A.2.*

```
2_incr_1( inout main_a, main_b; ) {
  main_a = main_a + 1;
  main_b = main_b + 1;
}

2_incr_2( inout main_c; ) {
  main_c = main_c + 1;
  main_c = main_c + 1;
}

main () {
  ...
  2_incr_1(a,b);
  2_incr_2(c);
  ...
}
```

This example illustrates how references must be disambiguated inside of a function. The behavior of the function depends of the context in which it is called. If we call `2_incr(&a,&b)` *where* `a` *and* `b` *are two separate variables, an implementation with two incrementers (possibly implemented in parallel as on Figure A.2a) is expected. When* `2_incr(&c,&c)` *is called the behavior becomes an incrementer by two (or two incrementers in serial as on Figure A.2b).*

Example A.3. *To reinforce the argument, another example is shown. Consider the function MAC is used to describe a multiply accumulate operator.*

```
void MAC(int *pa; int x; int c) {
  *pa = *pa + x*c;
}

void main( void ) {
  int a1, a2, x, c;
  ...
  MAC(&a1,x,c);
  ...
  MAC(&a2,x,c);
  ...
}
```

Assuming the same MAC operator is used for both calls, one possible implementation in HDL-like language could look like this.

```
MAC( in pa_tag;
     inout main_a1, main_a2;
     in x, c; ) {
  int star_pa[32], tmp_pa[32];

  temp = x * c;

  if(p_tag==0)
    star_pa = main_a1;
  else
    star_pa = main_a2;

  tmp_pa = star_pa + temp;

  if(p_tag==0)
    main_a1 = tmp_pa;
  else
    main_a2 = tmp_pb;
}

main {
...
MAC(0,a1,0,x,c);
...
MAC(1,0,a2,x,c);
...
}
```

The previous two examples illustrate how C functions can be synthesized. Functions that are not inlined get mapped to components. Since parameters are passed by reference using pointers, loads and stores are resolved within each function according to the different caller sites and the sharing information. Ports are added so that each component can access the difference locations that may be reference within the function. Pointers are encoded as in Chapter 4 and used to dynamically access these locations inside of the component.

A.2 Optimizing loads and stores inside of functions

The previous section presented how pointers may be resolved inside of functions. Case or branching statements are added in place of loads and stores in order to dynamically access the different locations the pointer may reference. In Chapter 6, compiler techniques are presented to optimize loads and stores. The goal of these optimizations is to reduce the number of registers used by reducing the number of live variables before loads and stores using temporary variables. These optimizations are effective when pointers may point to multiple variables stored in registers. This case happens mostly within functions. When functions are mapped to components shared by multiple call sites, the points-to set of the pointers consists of the different variables passed by reference at these difference call sites. The synthesis of functions with parameters passed by reference is therefore one of the main motivations for the two optimizations presented in Chapter 6. Examples A.4 and A.5 respectively illustrate the optimization of a loads and of a loads followed by stores within a function.

Example A.4. *Consider the following code segment:*

```
int MAC(int *p, int c, int x) {
  return *p+c*x;
}

main () {
  ...
  o1 = MAC(&a1, c, x1);
  ...
  o2 = MAC(&a2, c, x2);
}
```

Assuming that the function `MAC()` *is mapped to one shared component. One possible scheduled dataflow graph is presented on Figure A.3a. In such a case three registers are used at clocks 1 and 2 to store the values of* `p`, `a1` *and* `a2`. *When the load is optimized (i.e.*

star_p is defined as early as possible in the code), one register storing the value of `star_p` *is used instead of the three previous ones. The corresponding scheduled dataflow graph is shown on Figure A.3b.*

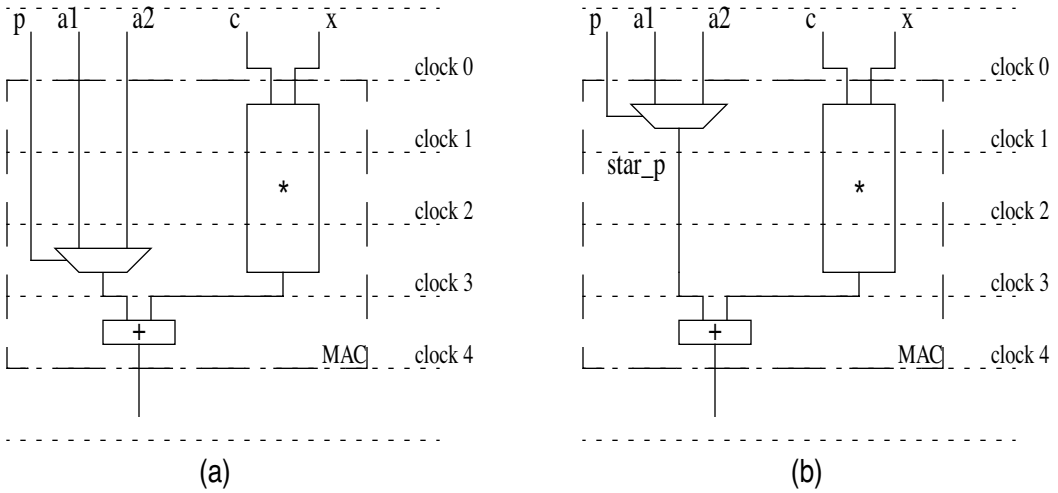


Figure A.3: synthesis of a function `MAC`(...) (a) without optimization and (b) with optimization

Note that in this example, the result of the optimization is similar to an ASAP scheduling of the dataflow graph. This is not always the case. An ASAP scheduling may only schedule the load operation (implemented as a mux) early on, which does not always minimize the storage.

Example A.5. *The following implementation of the* `MAC()` *operation includes a load followed by a store. In this case, temporary variables are used to save one register.*

```
void MAC(int *p, int c, int x) {
  *p = *p + c*x;
}

main() {
  ...
  MAC(&a1,c,x);
  ...
  MAC(&a2,c,y);
  ...
  MAC(&a3,c,z);
  ...
}
```

Assuming the function MAC is mapped to only one shared component, one possible scheduled-dataflow graph is shown on Figure A.4a. Note that we need 3 registers at clocks 1-4 to store the values of `a1`, `a2` *and* `a3`. *After optimizing the store, the temporary variables* `_star1_p` *and* `_star2_p` *are created and one register is saved as shown on Figure A.4_b.*

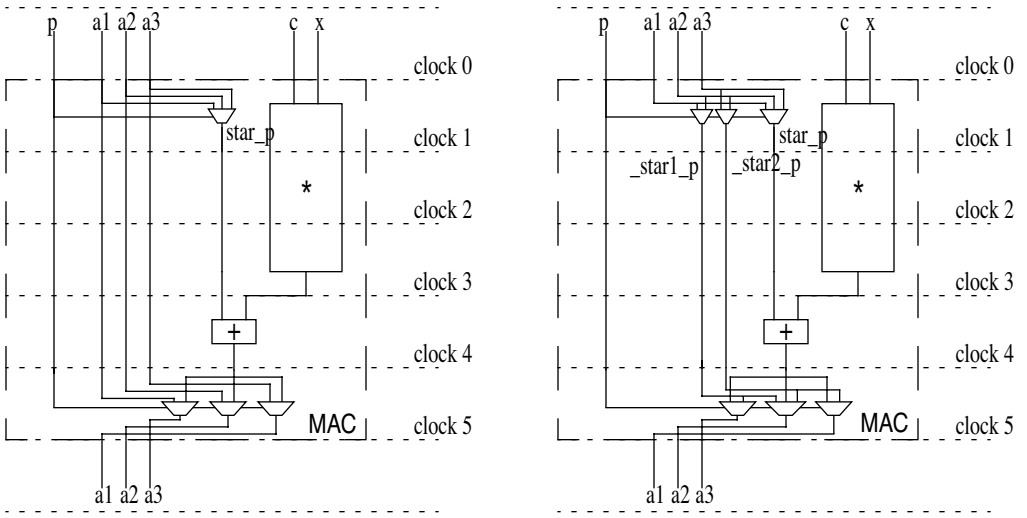


Figure A.4: synthesis of a function `MAC(...)` (a) without optimization and (b) with optimization

A.3 Encoding the value of the pointers inside of functions

Chapter 4 presents an hardware representation for the value of the pointers. In Chapter 7, a solution for encoding the pointers' values efficiently is presented. The goals of this encoding are two: 1) reducing the bit-widths of the pointers' tag (used to distinguish the location-sets referenced); 2) reducing the size of the circuits implementing assignments and comparisons of pointers. As we can see in Example A.6, when a function calls another function, the pointers in the callee are assigned by the caller. As a result, the sizes of the points-to sets increase with sharing. Synthesis of functions is therefore a great motivation for encoding pointers efficiently.

Example A.6. *Consider the following code segments, where functions* `foo()` *and* `bar()` *are each mapped to one component shared at the different call sites.*

```
main() {
  int a, b, c;
  ...
  foo(&a, &b, &c);
  ...
  foo(&b, &c, &d);
  ...
}

foo(int *r1, int *r2, int *r3) {
  ...
  bar(r1, r2);
  ...
  bar(r2, r3);
  ...
}

bar(int *q1, int *q2) {
...
}
```

Looking only at the initialization of the different pointers in the code, the behavior of the control logic generated looks like the following.

```
if(1st call of foo)
  { r1 = &a; r2 = &b; r3 = &c; }
else // (2nd call of foo)
  { r1 = &b; r2 = &c; r3 = &d; }

if(1st call of bar)
  { q1 = r1; q2 = r2; }
else // (2nd call of bar)
  { q1 = r2; q2 = r3; }
```

In this code we can see that pointer `r1` *inside of* `foo` *may point to* `a` *or* `b`*,* `r2` *may point to* `b` *or* `c`*, and* `r3` *may point to* `c` *or* `d`*. Then pointer* `q1`*, inside of function* `bar`*, may take the value of* `r1` *or* `r2`*, and* `q2` *may take the value of* `r2` *or* `r3`*. This corresponds exactly to the*

case presented in Example 7.1. The corresponding pointer-dependence graph and points-to sets for each pointer is shown again on Figure A.5. A solution to the problem of finding an optimal encoding of the pointers' value is presented in Example 7.6.

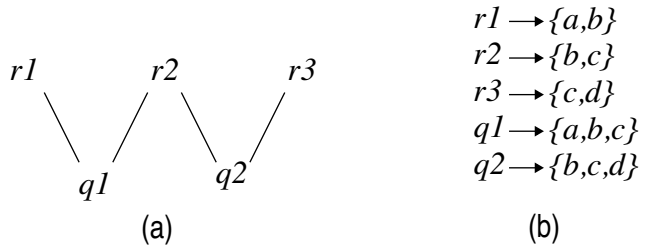


Figure A.5: (a) Pointer-dependence graph and (b) definitions of the points-to sets of each pointer

A.4 Summary

This sections describes how functions can be synthesized. The solution presented in this thesis supports parameters passed by reference and functions mapped to components. In C, pointers are used inside of functions to pass parameters by reference. During synthesis, the pointers inside of functions are resolved and encoded as described in Chapter 4. The points-to set of the pointers is defined by the pointer analysis but depends also on sharing information. The more sharing, the more locations the pointers may reference within a function. As a result, the synthesis of functions is a great opportunity for optimizing loads and stores as shown in Chapter 6. Moreover, function calls inside of functions requires assignments of pointers. The size of the circuits implementing such assignments can be also optimized using the encoding technique presented in Chapter 7.

In other languages such as Java or C++, parameters may directly be passed by reference without explicitly using pointers. The steps necessary for the synthesis of such reference involves however the same steps as the synthesis of pointers [33].

APPENDIX B. DISTANCE METRIC BETWEEN ENCODINGS OF POINTERS

Chapter 7 presented an algorithm to efficiently encode the symbols (i.e. references) in the points-to set of each pointer. A formulation of the encoding problem is presented in Section 7.2. Such formulation introduces the distance function $dist()$ between the two encoded sets. When the pointers have the same points-to set and the encoding has the same length n, $dist()$ is simply defined as:

$$dist(E_i, E_j) = \min_{perm()} \left(\sum_{k=1}^{N} H(perm(e_k^i), e_k^j) \right) \quad \text{(B.1)}$$

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

However, in general, the points-to sets may differ and their encoding may have different lengths (bit-widths). The computation of the distance is then more complex. This appendix presents a more general definition of the $dist()$ function.

Since the encodings may have different lengths, 0s are first added to the shorter encoding. Both encodings have then the same length. Note that padding with only 1s instead of 0s would lead to the same result.

If the points-to sets Π_i and Π_j are not the same (i.e., they do not contain the same set of symbols), we are only interested in the symbols in the intersection of the two points-to sets. Namely, let $K_{i,j}$ be the set of the indices of the symbols in points-to set Π_i that are also in points-to set Π_j.

$$K_{i,j} = \{k \text{ such that } s_k^i \in \Pi_i \cap \Pi_j\} \tag{B.2}$$

Since we do not consider all of the symbols in the points-to set we only want to compute the distance on the relevant bits in the encoding. As a result, we define a mask function in order to mask to non-relevant bits. For a set of symbols $\{s_k^i \text{ such that } k \in K\}$ the bit vector representing the non-constant bits in the encoding of these symbols is given by the bit-vector $mask(E_i, K)$ defined as follows:

$$mask(E_i, K) = \bigvee_{(k,l) \in K^2} e_k^i \oplus e_l^i \tag{B.3}$$

where the XOR operator $\oplus$ and the OR operator $\vee$ are applied bitwise to the bit vectors e_k^i and e_l^i.

Example B.1. *In the encoding shown in Figure 7.2a, the pointer* `r2` *may point to variables* `b` *or* `c`*, where* `b` *is associated with value 0 and* `c` *is assigned to value 1. Pointer* `q1` *may point to* `a`*,* `b`*, or* `c`*. However, in the encoding of* `q1`*,* `b` *is associated with value 01 and* `c` *is assigned to value 10. The intersection of the points-to sets of* `q1` *and* `r2` *is {*`b`*,*`c`*}. The*

mask for `r2` *is* $00 \oplus 01 = 01$ *after adding one 0 on the left to increase the code length to 2. Pointer* `q1` *has* $01 \oplus 10 = 11$ *as a mask.*

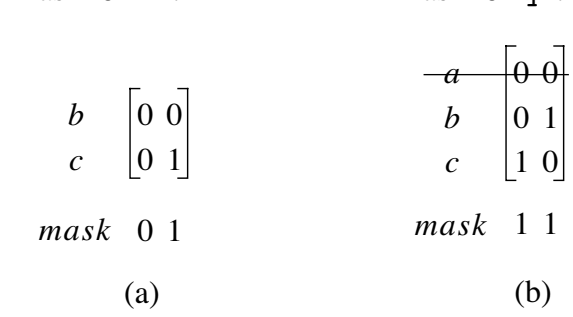


Figure B.1. Computation of the mask bit-vector for (a) `r2` and (b) `q1`.

On Figure B.1a, the first bit of the mask for `r2` *is 0 because the first column of the encoding matrix is constant equal to 0. The second bit of the mask is 1 because the values in the second column are 0 and 1. For the computation of the mask of q1 on Figure B.1b, the first row of the matrix, which corresponds to the encoding of symbol* `a`*, is discarded because it is not in the points-to set of r2. The resulting mask for q1 is 11.*

The distance is then computed as in Eq. B.1. Instead of computing the Hamming distance on all bits of the code, we only consider the non-constant bits for the symbols that are in the intersection of the two points-to set. The distance, in general, is defined as follows:

$$dist(E_i, E_j) =$$

$$\min_{perm()} \left(\sum_{\{(k,l) \text{ s.t. } (s_k^i = s_l^j)\}} H(perm(mask(E_i, K_{i,j}) \wedge e_k^i), mask(E_j, K_{j,i}) \wedge e_l^j) \right) \quad \text{(B.4)}$$

where $H()$ is the Hamming distance and the AND operation $\wedge$ is applied bitwise to the bit vector resulting of $mask(...)$ and e_k^i.

Example B.2. *For the encoding shown in Figure 7.2a, using the masks computed in Example B.1, the distance between the encodings of* `r2` *and* `q1` *is then:*

$$\min_{perm()} (H(perm(01 \wedge 00), 11 \wedge 01) + H(perm(01 \wedge 01), 11 \wedge 10)) =$$

$$H(00, 01) + H(10, 10) = 1 + 0 = 1 \quad \text{(B.5)}$$

In the encoding shown in Figure 7.2b, the encoding of `r2` *remains unchanged. On the other hand, for* `q1`, `b` *is associated with value 01 and* `c` *is assigned to value 11. The mask for* `q1` *is then* $01 \oplus 11 = 10$. *The distance between the two encodings is then*

$$\min_{perm()} (H(perm(01 \wedge 00), 10 \wedge 01) + H(perm(01 \wedge 01), 10 \wedge 11)) =$$

$$H(00, 00) + H(10, 10) = 0 + 0 = 0 \quad \text{(B.6)}$$

We can show in this case that Eq. 7.2 is then minimal equal to 0. Since Eq. 7.1 is also minimal too with this encoding (minimal length), we have an optimal encoding. The complexity of the circuit implementing the translation of the pointers' value in the assignment `r1=r2` *is then minimal.*

To summarize this appendix, a metric for computing the distance between two encoded points-to sets was presented. This metric effectively abstracts the complexity of the circuit implementing the translation of the values of the pointers. It handles the case of two different points-to sets encoded with different lengths. The idea was to pad the shorter code with 0s. Then, in the computation, only the encodings of the symbols that are common to the two points-to sets are considered and the unrelevent bits are masked out.