Seventh International Conference on Formal Methods in Computer-Aided Design

Global Optimization of Compositional Systems

Fadi Zaraket John Pape Adnan Aziz Magarida Jacome Sarfraz Khurshid
The University of Texas at Austin

***Abstract*—Embedded systems typically consist of a composition of a set of hardware and software IP modules. Each module is heavily optimized by itself. However, when these modules are composed together, significant additional opportunities for optimizations are introduced because only a subset of the entire functionality is actually used. We propose COSE—a technique to jointly optimize such designs. We use symbolic execution to compute invariants in each component of the design. We propagate these invariants as constraints to other modules using global flow analysis of the composition of the design. This captures optimizations that go beyond, and are qualitatively different than, those achievable by compiler optimization techniques such as common subexpression elimination, which are localized. We again employ static analysis techniques to perform optimizations subject to these constraints. We implemented COSE in the Metropolis platform and achieved significant optimizations using reasonable computational resources.**

I. Introduction

An embedded system performs one or more dedicated tasks using one or more processors that communicate with other dedicated devices, without being referred to as a computer [1], [2]. Embedded systems arise in many applications: examples include communications, image processing, and automotive electronics [2].

Embedded systems are heterogeneous by nature. For example, they combine application specific integrated circuits (ASICs) and field programmable gate arrays (FPGAs) with embedded software. A diversity of platforms try to employ and integrate different models of computation (MoC) to address the co-design problem of embedded systems. The platforms need to provide tools to specify, synthesize, and validate an embedded design.

Metropolis is an example of a design tool for embedded systems. The designer specifies the system using the Metropolis Metamodel (MMM) and Metropolis synthesizes the design by configuring parametrized architectural elements, dynamic scheduling algorithms, interface blocks, as well as final software and hardware implementation. Many techniques exist to optimize each of these components as well as the underlying communication network [3], [4], [5]. However, there is a need for developing new techniques that can perform optimizations across hierarchical boundaries.

We present *co-optimization using symbolic execution* (COSE), a novel technique that works across components of an embedded design to optimize structures therein. COSE automatically detects opportunities for optimization thereby avoiding labor on the part of the designer who would otherwise have to manually select the subset of functionality used. COSE detects invariants that the designer may not know or may not recognize as useful for optimizing other components. Furthermore, there is anecdotal evidence that having a designer manually simplify components using knowledge of their environments is a buggy process; we use a formal technique to compute opportunities for optimization, thereby guaranteeing the optimized and original designs are functionally equivalent.

Key to our approach is symbolic execution (SE) [6], [7]. We use SE to analyze software components and define a limited set of values that software feeds hardware as constraints. SE explores all possible paths of execution of the code specifying a component. It does this by accumulating *path conditions* (PCs) and annotating them to the corresponding segments of the component. A PC is associated with a branch of code and consists of the conjunction of conditions over input and state variables necessary and sufficient for the branch to execute. These PCs define constraints that limit the set of values that software feeds hardware. We propagate these constraints across the networks of the design and use *static analysis* techniques such as *constant propagation*, *redundancy removal*, and *don't care* optimizations to reduce the hardware components.

SE performs particularly well in the context of simple inequality checks and mapping assignments. This makes it suitable for our problem of detecting invariants that raise from different configurations and parameterizations of hardware components in software. COSE applies Juzi [6], an SE tool, to extract PCs for every line of code in an MMM component. Juzi uses CVC-lite [8] to solve or simplify PCs. CVC-Lite is a theorem prover for the Satisfiability Modulo Theory problem and it operates on logics that can express equality, inequality, and arithmetic operations.

We make the following key contributions:

1) We develop COSE—a technique that crosses software and hardware boundaries to perform joint optimizations.
2) We use SE to detect opportunities for optimization that are qualitatively different from those detectable with localized compiler optimizations and computationally infeasible with pure Boolean netlist analysis.
3) We implemented a prototype for COSE in the Metropolis framework and achieved significant results on two designs with significant complexity—a realtime image processing system and a switch fabric supporting mixed-mode traffic.

Our paper is structured as follows: we present a motivating example in Section II. In Section III-A we introduce Metropolis and the MMM. We illustrate SE with an example and elaborate on its advantages and limitations in Section III-B. In Section IV we present our prototype implementation. We present two case studies in Section V. We discuss related work in Section VI, and we finally conclude in Section VII.

We convey the key ideas behind COSE using examples and high-level description and rely on the Metropolis metamodel

0-7695-3023-0/07 $25.00 © 2007 IEEE
DOI 10.1109/.17

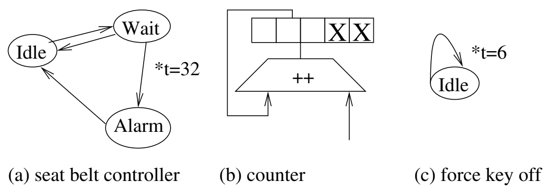


Fig. 1. Modified seat belt

to formalize the different techniques we use as well as their compatibility. We use the informal style for pedagogical reasons, as well as the 8 page submission limit.

II. Example

The state diagram in Figure 1(a) describes the classical seat belt controller of [9]. The controller waits for at most 32 seconds after the car key is turned on for the seat belt to be locked. If the seat belt was not locked, then the alarm will be on. Figure 1(b) shows the counter that implements the timer. Since it counts till 32, then at least 5 state bits are needed in the hardware implementation.

Suppose the car manufacturer decides to add a new safety feature and a designer implements it in a new component as shown in Figure 1(c). The new component monitors the same timer signal and forces the car to turn off if the seat belt was not locked before 6 seconds have elapsed. All we need to count for now is 6, and thus we only need 3 bits. However, we can see the potential for this reduction only when we co-optimize.

The same phenomenon is routinely faced in embedded system design. Examples include bus controllers with preemptive logic used in non-preemptive applications, cache IP that has pollution logic used in an application that never invalidates the data, an Internet Protocol processor that supports IPv6 used in a router that drops all IPv6 packets at the ingress, a microprocessor core that supports multiplication used in an application that is purely control dominated, etc.

III. Basic Concepts

In this chapter we review the Metropolis development environment. Metropolis provides a framework that integrates a set of tools to operate and manipulate a common design representation. We then introduce and illustrate the SE technique with an example.

A. Metropolis Overview

Metropolis provides a metamodel to support existing MoCs and accommodate new ones. Similar to other platforms like Polis [10], Ptolemy [11], SystemC [12], and SpecC [13]. Metropolis allows concurrent design of multiple processes and uses channels to communicate between them. It is different in that it introduces orthogonalities between (1.) computation versus communication, (2.) functionality versus architecture, and (3.) behavior versus performance. These processes are orthogonal since computation refinement is usually a manual process, functional and architectural specifications are decided by different groups, and performance constraints are specified by a group other than the behavior designers.

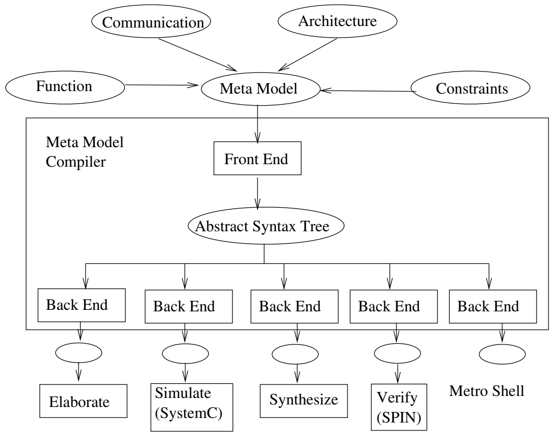


Fig. 2. Metropolis platform

The tools integrated in Metropolis allow designers to specify, partition, synthesize, and validate an embedded system. By partition we mean to decide which parts of the design get implemented by software that runs on programmable components and which parts get implemented directly in dedicated hardware devices. Synthesis compiles the software to optimized object code and hardware into optimized configurations of logic gate libraries. Validation checks whether the final implementation meets the properties in the specifications [14].

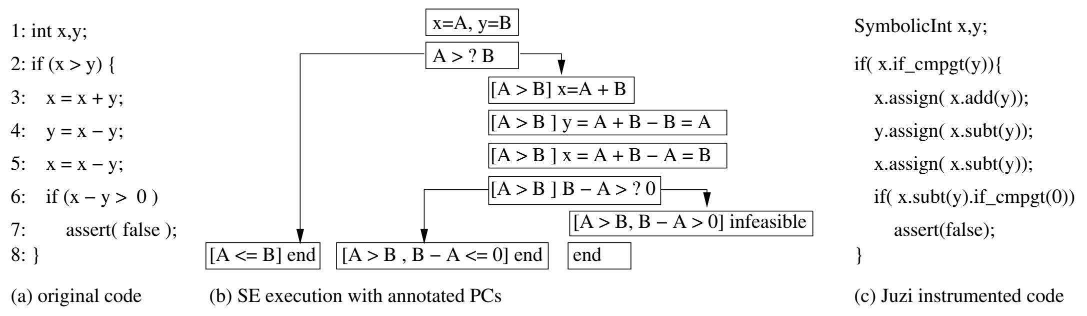
Metropolis uses MMM, an extension of the Java programming language with access restriction paradigms, as the design description language.

The diagram in Figure 2 shows the different components of the Metropolis platform. Metropolis provides several tools to manipulate the design. The two most commonly used are the Elaborator and the SystemC [12] code generator. The Elaborator parses the MMM specifications and generates an *abstract syntax tree* (AST). All the other tools in the Metropolis framework operate on the AST. The SystemC generator provides a path to simulation, verification, and synthesis. Metropolis also has another verification path to the SPIN [15] model checker.

1) **Design Methodology:** Metropolis allows a recursive design cycle with three phases. In Phase 1, the designer specifies the functional behavior of the design. This uses one or more MoC to describe (1.) the computation components, (2.) the communication components, (3.) the connections, and (4.) the hierarchy of the design. In MMM these are referred to as processes, media, interfaces and netlists respectively.

In Phase 2, the designer describes the target architecture platform of the design as a library of services. In Phase 3, the designer maps the functional specifications to the architecture services. In summary, the mapping is an intersection operation that constrains a functional interface to an architectural one.

The functional and architectural netlists may be developed by Metropolis designers and may also be provided as libraries. A platform vendor, for instance, may provide an architecture MMM of an FPGA to its customers. Similarly, a design team may develop a standard library of commonly used functional processes to promote reuse.



```
1: int x,y;
2: if (x > y) {
3:    x = x + y;
4:    y = x − y;
5:    x = x − y;
6:    if (x − y >  0 )
7:        assert( false );
8: }
```

```
SymbolicInt x,y;

if( x.if_cmpgt(y)){
    x.assign( x.add(y));
    y.assign( x.subt(y));
    x.assign( x.subt(y));
    if( x.subt(y).if_cmpgt(0))
        assert(false);
}
```

(a) original code (b) SE execution with annotated PCs (c) Juzi instrumented code

Fig. 3. Symbolic execution

2) **The Meta Model:** Metropolis uses a meta model to represent a design. Like the tagged-signal model [16], MMM is not tied to a particular MoC. MMM provides a set of computation and communication building blocks that can be customized to describe the semantics of many different MoCs thus enabling the modeling of heterogeneous systems.

MMM builds on the syntax and semantics of the Java programming language. It restricts itself to a subset of Java that includes inheritance, basic logical and arithmetic operators, loops, and conditionals. MMM introduces special class specifiers such as *process*, *media*, and *quantity* and supports method and field access control using modifiers like *eval*, *update*, and *port*. It also supports logical implication and equivalence operators in its embedded linear temporal logic (LTL) [17] used to specify constraints.

Processes are the active elements of the design and are typically used to perform computations via executing their thread methods. To decouple communication from computation, processes do not communicate directly with each other, they connect through interface ports and media. This allows the communication semantics to change without the need to change the processes.

Media are the communication components of the design and facilitate communication between processes and other media. They are passive, meaning they do not perform computation, but they may contain data and state.

The Metamodel supports two constraint formula types: LTL and logic of constraints (LOC). In addition, often LTL constraints form mappings expressing the equivalence between functional and architectural components.

A netlist connects the Metamodel components. Functional netlists usually consist of processes and media, while architectural netlists consist of processes, media, quantities, and constraints. Netlists may be connected hierarchically. A mapping netlist, for instance, can contain a functional netlist, an architectural netlist, and constraints between the two.

B. Symbolic Execution

SE is a technique that explores all possible executions of a program by traversing all possible branching conditions while bookkeeping symbolic values for all variables involved in the program at each atomic step. For every branch, SE adds the condition for the branch it takes to a PC in conjunction. Upon completion, all PCs are evaluated. Some of the accumulated PCs may be proved unsatisfiable, and thus the code these PCs annotate is proved dead since there is no setting for the program variables that satisfies its PC.

In Figure 3(b) we show the symbolic execution of the generic code in Figure 3(a). SE techniques start by initializing variables x and y to symbolic values A and B. At the first branch on line 2, the two choices are valid so they are both considered. The first, $A \leq B$, exits the block. The second, $A > B$, enters the body of the conditional statement. The values of x and y are updated at each line until the next branch is reached on line 6. At that point the first condition, $B - A \leq 0$, exits the block. The second, $B - A > 0$, leads to the assert statement. However the PC, accumulated from the branch on line 2, conflicts with the symbolic check on the values of x and y. Thus we can conclude that the assert statement is not accessible.

This shows how the SE technique can effectively detect dead code in control and data flow dominated circuits. With SE we can also annotate each line of code with all the PCs that allow the line to be executed. The tool Juzi [6] implements SE for Java code. It does so by instrumenting the Java-byte code of the Java class in question to use symbolic implementations for its variables. Figure 3(c) shows the Java code equivalent to the automatic Juzi instrumentation for the code in Figure 3(a). The variables x and y are declared as `SymbolicInt` variables and the arithmetic and Boolean operations and conditions applied to them are replaced by symbolic operations. The symbolic `assign`, `add`, and `subt` operations update the correspondence between x and y and the symbolic values A and B respectively. The `if_cmpgt` updates and checks the PC with the different choices it makes. Juzi also introduces *lazy initialization* where it initializes a variable only when it is actually used rather than when it is declared. Juzi supports dynamically allocated structures and data structures with self-loops – e.g., linked lists, and respects method preconditions.

Under the hood, Juzi uses CVC-lite to decide a PC or to simplify it into an equivalent more compact PC. CVC-lite is a validity checker for typed first order logic with interpreted theories. It supports integer and real arithmetic, equality and inequalities, predicate subtypes, partial functions, uninterpreted functions, arrays, records, bitvectors, and quantifiers [8]. COSE leverages these capabilities to work on a high level of abstraction and detect invariants that are qualitatively different than those detected at the Boolean level.

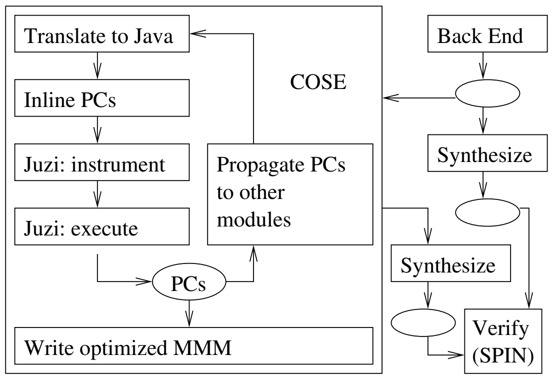


Fig. 4. COSE in Metropolis

IV. Our Approach

Before we proceed to present the technical solution, we discuss two fundamental questions. First, one may question whether software can be developed before hardware is committed. Second, new versions of software may try to use hardware that was optimized away.

The software dependence on ready hardware question is methodology specific. The MMM assumes a complete design at every level of abstraction, and thus the reductions obtained by co-optimization apply only to that level. At any refinement, reductions need to be discarded and co-optimization can be applied again. The flexibility question is inherent in all models and is specific to the co-optimization technique. COSE provides the sets of constraints it detected and used to optimize other components. Thus it allows the designers to either stay within the framework of the optimizations or to discard the optimizations and run COSE again in case new conflicting constraints were found.

A. COSE Overview

Figure 4 shows how COSE is integrated into the Metropolis framework. COSE is an iterative process that uses SE and static analysis techniques to optimize a design at hand. It takes as input an MMM netlist Φ, and a starting PC σ that could be empty. It selects one interesting process Π in Φ, and it uses Juzi to symbolically execute the thread function in Π, with σ as the starting PC.

COSE uses Juzi and CVC-lite to perform SE. This is convenient since MMM is an extension to Java with access restrictions. COSE translates Π to Java and calls it Π_{Java}. The translation to Java is straightforward and works with inlining method calls and substituting ports by variables with Boolean validity guards to control read and write operations and allow a run-time protection mechanism. Then COSE compiles Π_{Java} into byte-code, Π_{BC}, and feeds that to the Juzi instrumentor class. COSE then symbolically executes the instrumented class and gathers the generated PCs.

COSE follows the usage of ports in the process and joins all the path conditions annotated to the code of the ports in disjunction into σ_i^{port}. Then each σ_i^{port} is treated as a constraint to the port and is propagated through the corresponding connections in Φ. COSE repeats these steps for every medium and process that connects to Π through a constrained port.

B. COSE Optimizations

We annotate each line of MMM code with the disjunction of the PCs generated for it from Juzi. We resolve and compact the PCs using CVC-lite. A PC that evaluates to false is *infeasible* and the code it annotates is dead-code and thus can be optimized away.

COSE can also perform range restriction optimizations. If, for example, COSE detects PCs restricting all indexes reading and writing into an array to a specific range, it can resize the array and use a smaller number of bits for the indexes.

COSE also checks for mutually exclusive blocks of code via evaluating the PCs of interesting blocks of code that use the same ports. Once two mutually exclusive blocks of code are detected, COSE adds a constraint to the netlist connecting the two blocks of code to declare they are mutually exclusive. Synthesis tools can use this to optimize communication between ports depending on the underlying MoC [18].

COSE generates optimized MMM modules that are annotated with detected and simplified PCs. In addition to conditional statements, COSE embeds the PCs it detected on ports as LTL constraints in the MMM module describing the netlist connections of the design. We pass the optimized and constrained MMM to a synthesis tool where we target reductions achieved by static analysis such as constant propagation, redundancy removal [19], [20], and observability don't care optimizations [21].

COSE operates only on the MMM level. It uses the netlist connecting the different components of a design only to follow the connections regardless of the semantics of the underlying MoC. Thus the optimizations and constraints computed by COSE are valid regardless of the underlying MoC.

C. Dependency Map and Constraint Propagation

COSE propagates constraints through ports. A port has an interface type and allows access to methods declared in the interface. More than one process may own ports with the same interface type, and media implement the interface methods. COSE follows the port declarations and usages in the entire netlist Φ and forms a dependency graph. The nodes of the dependency graph are processes, media, and interfaces. The arcs are connected ports. All the connections in MMM are done with the connect keyword which facilitates building such a map. Connections in MMM have to be resolved statically and are done at the netlist level. This helps run SE on the component level only and helps COSE scale to large compositions. A PC is coded into a conditional constraint and is annotated either to the thread method of a process, or to the implementation of an interface method in a media.

D. Synthesis Reduction Techniques

COSE can use the native synthesis tools in Metropolis or can run public domain synthesis optimization tools like SIS [21] to optimize the components with the computed

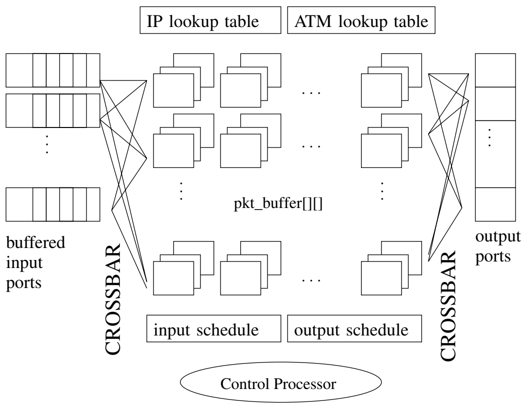


Fig. 5. Switch fabric design

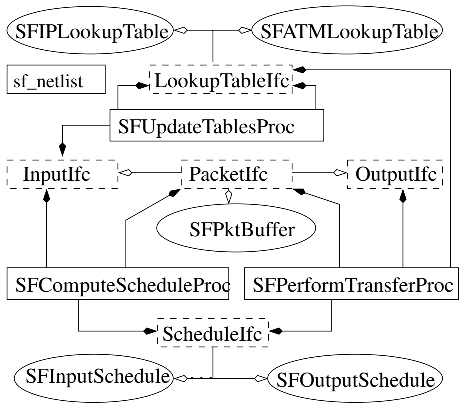


Fig. 6. Class diagram for switch fabric metamodel

constraints. We obtain the best optimization gains when COSE detects dead code that instantiates or activates other components of the netlist. Constraint based optimizations intersect with sequential don't care reductions. As previously stated, COSE has the advantage of detecting invariants on a level of abstraction higher than classical don't care optimization techniques which use Boolean constraints. This is mainly because COSE uses SE which produces PC expressed with equality, inequality and arithmetic logic.

E. Illustration on the Seat Belt Example

For the modified seat belt example described in Section II, in a single iteration COSE propagated a PC that constrains the counter port to ≤ 6 in the timer. If the timer is implemented in hardware, this is enough to set the two most significant state bits of the counter to 0. Constant propagation techniques can propagate the 0 values and drop the constant state bits. Juzi instrumented and symbolically executed the seat belt example in less than 5 seconds.

V. Case Studies

In this section, we apply COSE to reduce two designs with significant complexity. The first is a switch fabric for mixed IP and ATM traffic that can be deployed in several configurations such as an IP-only traffic application. The second is an image processing system that identifies objects in real time pictures and it also can be used in different configuration such as military usages and home video surveillance.

A. Switch Fabric

In Figure 5 we show the organization of a shared-memory switch fabric [22] for mixed ATM and IP traffic. The switch fabric receives packets of data on its input ports and correctly forwards them to their destination ports. A packet can be described as a data structure composed of a protocol field, a class field, a destination address and a payload stream of data. Input ports are modeled by queued buffers of packets. Output ports are assumed to hold the data for immediate consumption.

When two packets arrive which need to be forwarded to the same output port, one is stored in the packet buffer. Pooling the packet memory leads to better buffer utilization [23], and high-end core routers, such as the Juniper M-series are organized in this fashion [24].

The switch fabric in Figure 5 supports both IP and ATM traffic. It implements class-of-service: each packet has a class-of-service field which identifies its priority. The control processor computes two schedules, taking into account the free packet buffers: the *input schedule* maps input ports to available packet buffers, and the *output schedule* maps valid packet buffers to output ports. The physical transfer of packets is performed via the crossbars after the schedule is computed.

The design is configured using in-band signaling through IP and ATM packets specially marked with a control code in the first byte of the packet payload. A control packet contains a destination and a port number and instructs the switch fabric node to update the forwarding table to map the destination to the output port. The design in Figure 5 implements this via two lookup tables for the protocols it supports. The lookup tables are used later to map data packets with destination addresses to the correct ports.

1) **MMM for Switch Fabric:** Figure 6 shows the MMM class diagram of the switch fabric design. Boxes denote process classes, dotted boxes denote interfaces, ellipses show media classes, and filled and empty arrows denote ownership and inheritance relations respectively. The design is connected through the netlist object sf_netlist. The input and output schedules are media objects that implement the ScheduleIfc interface. The IP and ATM lookup tables are media objects that implement the LookupTableIfc interface. The internal memory is a two dimensional array of the SFPktBuffer media which implements the PacketIfc interface. The sf_netlist has input and output port interfaces that also implement the PacketIfc interface. The computational elements of the switch fabric design are expressed in three processes. The SFUpdateTablesProc process receives control input packets and updates the lookup tables. SFComputeScheduleProc looks at the input and output ports and the internal memory through PacketIfc and updates the input and output schedules. SFPerformTransferProc is the third process and it actually reads the schedules and performs the data transfer accordingly.

2) Results for Switch Fabric: We assumed the deployment of the switch fabric design in Figure 5 in multiple scenarios. The first scenario is with network elements that produce IP only traffic. The second scenario was with local network elements that are limited to the use of few output ports.

We modeled the IP-only traffic by a CreateInputs process. The following fragment of pseudo code from CreateInputs shows how the protocol field is set to IP_PROTOCOL for valid inputs.

```
int inIdx;
for ( inIdx = 0 ; inIdx < NUM_IN; inIdx++ ) {
 inBuff[inIdx].valid = chooseBoolean();

 if(inBuff[inIdx].valid == 1){
  // if (chooseBoolean() == 1)
     inBuff[inIdx].pkt.protocol = IP_PROTOCOL;
  //else
  // inBuff[inIdx].pkt.protocol = ATM_PROTOCOL;
  ...  }
```

Similar to Figure 3, COSE instruments all fields of the pkt as SymbolicInt and thus detects the path condition PC_{prtcl}, inBuff[inIdx].protocol == IP_PROTOCOL, in all execution branches of the CreateInputs process. COSE propagates PC_{prtcl} to SFComputeSchedule through the InputIfc ports. COSE was able to determine that all blocks of code conditioned with if(inBuff[inIdx].protocol == ATM_PROTOCOL) are infeasible in all branches of execution of SFComputeSchedule. COSE annotates all these blocks of code with an if(false) PC and thus optimizes them out. In addition, COSE annotated the PacketIfc and the ScheduleIfc interfaces with PC_{prtcl} since it still holds across all execution branches in SFComputeSchedule. Then in turn, COSE propagated the PC_{prtcl} to the SFPerformTransferProc process. All blocks of code therein conditioned with if(inBuff[inIdx].protocol == ATM_PROTOCOL) were optimized out.

With 4 input ports, 4 output ports, and an 8×16 internal packet buffer, it took COSE 3 hours and 12 minutes to complete this optimization with $36,816$ symbolic integer variables and more than 2 million branches of execution.

We modeled the network with limited output port accessibility with a CreateInputs process that sends control packets with a limited destination port value.

```
int inIdx;
for(inIdx = 0; i < NUM_IN; i++){
  inBuff[inIdx].valid = chooseBoolean();
  if(inBuff[inIdx].valid == 1){
    int choiceDataVsControl = chooseBoolean();
    if(choiceDataVsCntrl == 1){
      inBuff[inIdx].pkt.data[0] = CTRL_PKT;
      inBuff[inIdx].pkt.data[1] =
        makeDestIPAddress();
      inBuff[inIdx].pkt.data[2] =
        chooseInt().remainder(4);
      ...
    } else { // setup data packet
      ...} } }
```

COSE was able to detect the path condition PC_{port}, (inBuff[inIdx].pkt. data[0] == CTRL_PKT) AND (inBuff[inIdx].pkt. data[2] ¡ 4). COSE propagated the PC_{port} condition, through the InputIfc interface, to SFUpdateTablesProc and then annotated all the LookupTableIfc interfaces with lookupTable[tableIdx].outputPort ¡ 4. COSE propagated the latter PC to SFPerformTransferProc and annotated the only block of code that writes to the output ports with it.

```
if(outPortIdx < 4){ // COSE ANNOTATION
 outBuff.[outPortIdx].valid = 1;
 outBuff.[outPortIdx].pkt = getPcktFromScheduleAndMemory();
  ...  } // END COSE ANNOTATION
```

With this annotation COSE enabled the synthesizer to drop the 4 output ports that are never written to or validated. COSE ran with $61,018$ symbolic integer variables and took 4 hours and 7 minutes to inspect all feasible branches of execution and compute this optimization.

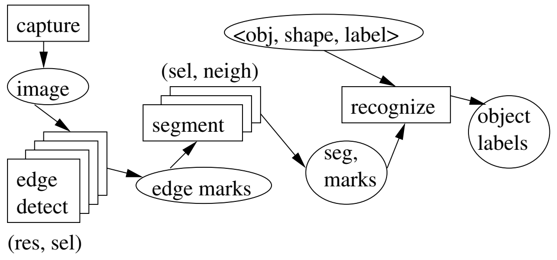


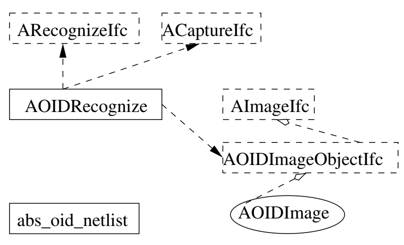
Fig. 7. Flow diagram for the ObjectID application

B. Case Study: Video Surveillance

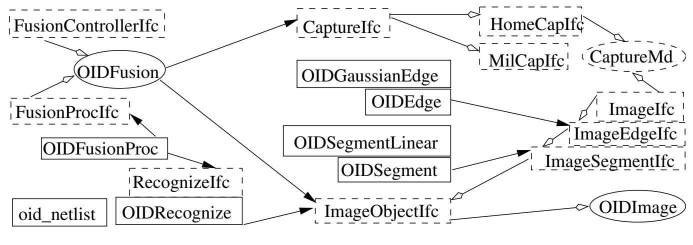
Figure 7 shows an image processing video surveillance application, also called object identification, that was developed in an MS project [25]. The system accepts an image and produces labels of the objects with their coordinates in the image. One or more edge detectors scan the image and produce edge marks that are visited by one or more segmentation algorithms to produce segmentation marks. The segmentation marks partition the image with the borders of the main shapes in it. Then a pattern recognition algorithm is used to match the shapes in the segmented image to a library of objects. The application instantiates 4 edge detectors and 3 segmentors and uses them depending on the resolution of the requested edge and recognition. We assume the system was designed for military usages that need high resolution analysis, and then it is used for home surveillance purposes where lower resolutions are suitable. If only home surveillance resolution is desired, just two detectors and one segmentor are needed.

1) MMM Model for ObjectId: We implemented the object identification example in MMM. The system originally took 16 weeks of work by one designer and is around 4000 lines of C and RTL code. We modeled it with around 1255 lines of MMM code. The diagram in Figure 8(a) shows the high abstract view of the ObjectID application, and the diagram in Figure 8(b) shows the detailed class diagram.

The OIDImage media implements OIDImageObjectIfc which defines interfaces for object identification and inherits image, edge, and segment interfaces. An OIDEdge process accesses the OIDImage media through owning a port of type ImageEdgeIfc. OIDGaussianEdge extends the OIDEdge process and implements one type of edge detectors which uses a Gaussian mask to differentiate pixels. We only named one type of edge detectors for brevity. Other well known techniques for edge detection such as the Sobel, Prewitt, Laplacian of Guassian (LoG), and the Canny detectors [26] may be used. Similarly to OIDEdge, OIDSegment owns a port of type



(a) abstract class diagram of ObjectID



(b) detailed implemetation class diagram of ObjectID

Fig. 8. Levels of abstraction of the Object ID application

`ImageSegmentIfc`, and `OIDSegmentLinear` implements one type of segmentation algorithms. The `OIDFusion` medium owns a `CaptureIfc`, and an `ImageObjectIfc`. It also implements the `FusionControllerIfc` and `FusionProcIfc` interfaces. `OIDFusionProc` owns `FusionProcIfc` and `RecognizeIfc` interfaces. The `OIDRecognize` process also owns similar interfaces.

The netlist `oid_netlist` instantiates 4 `OIDGaussionEdge` processes, 2 `OIDSegmentLinear` processes, one `OIDImage` medium, one `OIDFusion` medium, and one `OIDFusionProc` process. It also connects them appropriately through their owned and implemented ports. `CaptureIfc` and the object part of `ImageObjectIfc` are open to the outside world where the system can accept controlled input.

2) **ObjectID Dependency Map:** Figure 9 shows the code of the ObjectID netlist, `OIDFusion`, and `OIDGaussianEdge`. The arrow on the `connect` method call in the netlist constructor links the usage of the oid_fusion_ifc port to its implementation in `OIDFusion`. If the capture and the recognition interfaces happen to be set to 120 and 105 then the body of the `thread` method of the OIDGaussianEdge detectors instantiated as edge3 and edge4 will be dead code since the first statement in the condition is going to be always false. The arrows in the figure show how the dependency map was built from instantiations and connections in the netlist and usages of ports in process classes.

3) **COSE ObjectID Results:** In the ObjectID, we assumed a capture medium implementation that requires a low resolution. We were able to detect infeasible PCs on lines that activate the two high resolution edge detectors in one COSE iteration. In the second iteration and after propagating the infeasible PCs we were also able to detect dead code for the lines that activate one of the segmentation processes. We were able to reduce the system by 2 edge detector components and one segmentation component. It took COSE 15 minutes to instrument and symbolically execute the ObjectID code.

VI. Related Work

Compared to sequential optimization techniques from logic synthesis, specifically, those based on sequential don't cares [27], COSE operates at much higher level of abstraction than Boolean netlists—we treat high-level constructs such as integer variables, memories, etc., natively. We also make use of the decomposition of the design into modules. This leads to more compact models, reducing run-time complexity which has been a major drawback to sequential logic synthesis.

Analyzing software components with Boolean methods is far beyond the reach of current tools—there is too much state associated with the instruction set. We have in the past experimented with verifying a switch fabric with VIS [28], We implemented the scheduling in hardware, and it was considerably simpler than the scheduler employed in this paper. However, verification still suffered from BDD-blowup, and we were unable to verify elementary properties of a 2-input 2-output shared memory switch with 4 packets of buffering, no control packets, and no support for mixed-mode traffic.

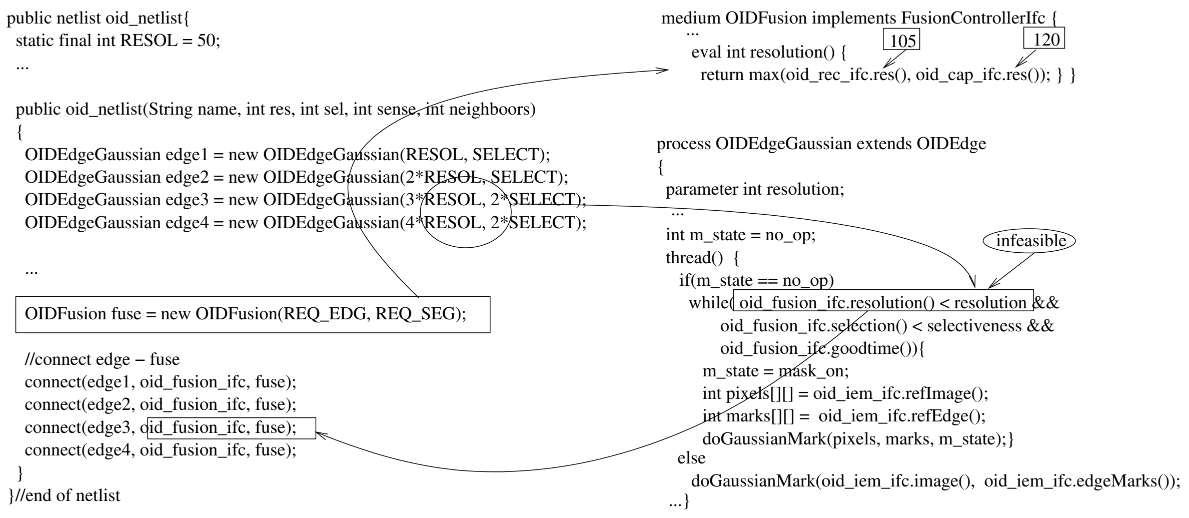
Our SE technique is different from symbolic reachability analysis [28] in many aspects. First, it does not examine the transition system of a component in whole and rather looks at the transition of interleaving variables that define the state space via executing the code line by line. Also it only considers reachable states via pruning infeasible PCs.

Prior dead-code detection techniques [29], [30] make use of conservative static analysis and can therefore erroneously classify paths as alive; in contrast, SE coupled with a decision procedure gives more accurate results for programs with bounded loops and recursion.

Similar to VeriSoft [31], COSE performs a *state-less* search: it systematically explores an *execution tree*, which represents all execution paths up to a desired bound, and backtracks using re-execution, ensuring that all (feasible) paths are explored. This contrasts with the *state-full* search of SPIN [15], which stores states using hashing. COSEs use of symbolic execution also contrasts with VeriSoft and SPIN, who do not use off-the-shelf decision procedures and require the user to a priori provide (tight) bounds on integer inputs.

VII. Summary

We introduced the use of SE to co-optimize embedded systems, presented COSE and implemented a prototype that uses Java inlining. We integrated COSE in the Metropolis framework and showed it can lead to significant optimizations on representative examples. COSE is a co-optimization mechanism that can find constraints in software and effectively propagate them to hardware. These constraints may not be observable by designers and lead to optimizations that are qualitatively different than reductions possible with localized compiler optimizations and Boolean level transformations.



```
public netlist oid_netlist{
  static final int RESOL = 50;
  ...

  public oid_netlist(String name, int res, int sel, int sense, int neighboors)
  {
    OIDEdgeGaussian edge1 = new OIDEdgeGaussian(RESOL, SELECT);
    OIDEdgeGaussian edge2 = new OIDEdgeGaussian(2*RESOL, SELECT);
    OIDEdgeGaussian edge3 = new OIDEdgeGaussian(3*RESOL, 2*SELECT);
    OIDEdgeGaussian edge4 = new OIDEdgeGaussian(4*RESOL, 2*SELECT);

    ...

    OIDFusion fuse = new OIDFusion(REQ_EDG, REQ_SEG);

    //connect edge − fuse
    connect(edge1, oid_fusion_ifc, fuse);
    connect(edge2, oid_fusion_ifc, fuse);
    connect(edge3, oid_fusion_ifc, fuse);
    connect(edge4, oid_fusion_ifc, fuse);
  }
}//end of netlist
```

```
medium OIDFusion implements FusionControllerIfc {
    ...
    eval int resolution() {
      return max(oid_rec_ifc.res(), oid_cap_ifc.res()); } }

process OIDEdgeGaussian extends OIDEdge
{
  parameter int resolution;
  ...
  int m_state = no_op;
  thread() {
    if(m_state == no_op)
      while( oid_fusion_ifc.resolution() < resolution &&
           oid_fusion_ifc.selection() < selectiveness &&
           oid_fusion_ifc.goodtime()){
        m_state = mask_on;
        int pixels[][] = oid_iem_ifc.refImage();
        int marks[][] =  oid_iem_ifc.refEdge();
        doGaussianMark(pixels, marks, m_state);}
    else
      doGaussianMark(oid_iem_ifc.image(),  oid_iem_ifc.edgeMarks());
 ...}
```

Fig. 9. Dependency map of ObjectID

Given the different levels of abstractions possible in Metropolis, COSE can accompany the concretization process one step at a time. COSE guarantees that the result of its optimization is functionally equivalent to the original design. In the future, we plan to optimize COSE via integrating it with a native difference equation solver instead of calling CVC-Lite. We are also exploring the use of SE to optimize linking programs across compilable software modules.

REFERENCES

[1] F. Balarin, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, M. Sgroi, and Y. Watanabe, "Modeling and designing heterogeneous systems," tech. rep., Cadence Berkeley Laboratories, Nov 2001.

[2] F. Balarin et al., "Metropolis: an integrated electronic system design environment," in *Computer 36*, Apr 2003.

[3] M. Chiodo, "Optimization and synthesis for complex reactive embedded systems by incremental collapsing," in *International symposium on Hardware/software codesign*, 2002.

[4] M. Chiodo, P. Guisto, A. Jurecska, L. Lavagno, H. Hsieh, K. Suzuki, A. L. Sangiovanni-Vincentelli, and E. Sentovich, "Synthesis of software programs for embedded control applications," in *Design Automation Conference*, 1995.

[5] M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavango, and A. Sangiovanni-Vincentelli, "Synthesis of mixed software-hardware implementation from CFSM specifications," in *International Workshop on Hardware-Software Codesign*, 1993.

[6] S. Khurshid, C. Pasareanu, and W. Visser, "Generalized symbolic execution for model checking and testing," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Apr 2003.

[7] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, 1976.

[8] C. Barrett and S. Berezin, "Cvc lite: A new implementation of the cooperating validity checker," in *Computer Aided Verification*, April 2004.

[9] P. S. Roop, A. Sowmya, and S. Ramesh, "k-time forced simulation: A formal verification technique for ip reuse," *International Conference on Computer Design*, 2002.

[10] F. Balarin et al., *Hardware-software codesign of embedded systems: The POLIS approach.* Academic Publishers, Boston, 1997.

[11] E. A. Lee, "Overview of the ptolemy project," tech. rep., University of California, Berkeley, Jul 2003.

[12] T. Grotker, S. Liao, G. Martin, and S. Swan, *System Design with SystemC.* Kluwer Academic Publishers, 2002.

[13] M. Fujita and H. Nakamura, "The standard specc language," in *International Symposium on Systems Synthesis*, 2001.

[14] F. Xie, G. Yang, and X. Song, "Component-based hardware/software co-verification," in *Formal Methods and Models for Co-Design*, July 2006.

[15] G. Holzmann, "The model checker SPIN," in *IEEE Transactions on Software Engineering*, May 1997.

[16] S. Edwards, L. Lavagno, E. A. Lee, and A. Sangiovanni-Vincentelli, "Design of embedded systems: Formal models, validation, and synthesis," *Proc. of the IEEE*, vol. 85, 1997.

[17] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper, "Simple on-the-fly automatic verification of linear temporal logic," in *International Symposium on Protocol Specification, Testing and Verification*, 1996.

[18] O. Bringmann, W. Rosenstiel, and A. Siebenborn, "Conflict analysis in multiprocess synthesis for optimized system integration," in *International conference on Hardware/software codesign and system synthesis*, 2005.

[19] H. Mony et al., "Scalable automated verification via expert-system guided transformations," in *Formal Methods in Computer-Aided Design*, Nov. 04.

[20] P. Bjesse and K. Claessen, "SAT-based verification without state space traversal," in *Formal Methods in Computer-Aided Design*, November 2000.

[21] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, "SIS: A System for Sequential Circuit Synthesis," tech. rep., Univ. of California, Berkeley, May 1992.

[22] J. Turner and N. Yamanaka, "Architectural Choices in Large Scale ATM Switches," *IEICE Transactions*, 1998.

[23] A. Prakash, A. Aziz, and V. Ramachandran, "Randomized Parallel Schedulers for Switch-Memory-Switch Routers: Analysis and Numerical Studies," in *IEEE Infocom*, 2004.

[24] J. Networks, "High speed switching device." US Patent 5,905,726, 1999.

[25] F. Zaraket, "Optimal Fusion and Objective Comparison of Edge Detectors ," Master's thesis, American University of Beirut, 2001.

[26] D. Marr and E. Hildreth, "Theory of edge detection," in *the Royal Society of London*, 1980.

[27] A. Aziz, F. Balarin, R. Brayton, and A. Sangiovanni-Vincentelli, "Sequential Synthesis Using `SIS`," *IEEE Trans. Computer-Aided Design*, vol. 19, Oct. 2000.

[28] R. Brayton, G. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. Ranjan, S. Sarwary, T. Shiple, G. Swamy, and T. Villa, "VIS," Nov. 1996.

[29] R. Bodek and R. Gupta, "Partial dead code elimination using slicing transformations," in *ACM Conference on Programming language design and implementation*, 1997.

[30] Y. A. Liu and S. D. Stoller, "Eliminating dead code on recursive data," in *Static Analysis Symposium*, 1999.

[31] S. Chandra, P. Godefroid, and C. Palm, "Software model checking in practice: an industrial case study," in *International Conference on Software Engineering*, 2002.