Automatic Extraction of Function Bodies from Software Binaries

Abstract

Migration of software, from older general purpose embedded processors onto newer mixed hardware/software platforms, is becoming an increasingly important topic. Generation of these designs requires partitioning of the original code. This partitioning can be done at several levels of granularity, a popular one being functions. In most cases, the partitioning of software is complicated due to the lack of function boundary information. To address this problem, this paper describes a method of automatically extracting function bodies from linked software binaries. It utilizes procedure-calling conventions along with limited control and data flow information. We use binaries of a representative DSP processor platform, the Texas Instruments C6000. Results are reported on eight benchmarks for which our algorithm successfully identifies all functions. This corresponds to 198% more functions than the number correctly identified using simple procedure calling conventions.

Key words: binary translation, procedure extraction, calling convention, hardware/software partitioning.

1 Introduction

Recent advances in embedded communications and control systems for personal and vehicular environments are driving efficient hardware and software implementations of complete systems-on-chip (SOC) designs. These applications require digital signal processing (DSP) functions that are typically mapped onto general-purpose DSP processors, such as the Texas Instruments TMS320C6000 and the Analog Devices SHARC processor. However, it is widely believed that such processors will be unable to support the computational requirements of future DSP applications. Hence, there is an increasing need to migrate these tasks into application-specific hardware components. To achieve this migration automatically, it is desirable to identify fragments of code or some functionality that would be better suited to a hardware implementation. This process of identification is known as hardware/software partitioning. Although partitioning can be performed at various levels of granularity (instructions, functions, kernels, etc.), we concentrate on the partition granularity of functions because of their wide usage.

This paper deals with the design of a compiler that automatically translates binaries targeted for general DSP processors into Register Transfer Level (RTL) VHDL or Verilog code to be mapped onto commercial FPGAs. A key motivation for developing a translator from assembly code or binary to hardware is that there is a large established code base of DSP algorithms optimized for specific processor families. Some of these algorithms are hand-coded for better performance. A second motivation is that tools are available to implement C/C++, MATLAB and SIMULINK designs on DSP processors. Therefore, the binaries generated by these tools might be an attractive alternative as an intermediate language. The close-to-hardware nature of assembly might also allow for better HW/SW partitioning decisions.

A key problem in this compilation process is the automatic recognition of procedure calls and code bodies from software binaries. This recovery of procedure calls and code bodies from software binaries can be considered to be a part of the decompilation process. It is also useful during binary translation since it is usually desirable to conform to the calling convention used by a compiler writer for the target architecture. Adherence to the Application Binary Interface of the target’s operating system allows the generated code to interact with other software. Procedure extraction might also be useful when analyzing binary code for security purposes, e.g. to weed out malicious code fragments.

The remainder of the paper is organized as follows. Section 2 reports on related work. An overview of the compiler framework for binary to hardware translation is presented in Section 3. A motivational problem is described in Section 4. The automated function extraction algorithm is described in Section 5. Section 6 analyzes the complexity of the algorithm and Section 7 reports experimental results on 8 benchmarks.

2 Related Work

There has been some related work in the field of binary translation, in converting assembly or binary code written for one processor to another processor’s instruction set architecture (ISA). The Transmeta Crusoe processor performs dynamic code translation on the fly using a technique called CodeMorphing by translating code from
an Intel x86 ISA and targeting the Crusoe processor, a
VLIW machine [15]. Cifuentes et al [5][6] present a
detailed analysis of different translation strategies. Bala et
al [16] have developed the Dynamo system for dynamic
binary optimization for the HP architecture. Gschwind [17]
has developed a similar system called BOA for the
PowerPC architecture. The major difference between these
and our paper is that we are making the translation from a
certain binary to hardware. Our target being hardware
allows us to be more flexible in terms of our target, but
introduces several new research questions.

Cifuentes et al [14] have reported algorithms for
identifying function calls from assembly programs using
predefined procedure call interfaces. Although this is an
attractive solution to extract function bodies, it is not
applicable to most DSP binaries. Cooper et al [11], on the
other hand, have reported methods to construct Control and
Data Flow Graphs from scheduled assembly code.
However, these methods cannot be used to extract function
bodies without significant modification.

Baily and Davidson [12] introduced a formal model to
specify procedure-calling conventions. Cifuentes and
Simon describe a procedure abstraction language that can
be used to specify the calling conventions for different
architectures [14]. They also discuss how use-def analysis
can be used for identification of function arguments. Mike
Van Emmerik describes the use of patterns to identify
library functions in executables [13]. This work helps in
identifying function calls but does not prove sufficient to
define complete function bodies. The calling conventions
help in identifying caller/callee prologues and epilogues. In
hand-written and/or optimized assembly it is possible that
the code comprising these conventions has been moved.
If functions pointers are passed as arguments to other
functions, it is very likely to miss the called functions
completely. In the course of migrating optimized code for
an echo canceling application (from TI C6211) both
scenarios were encountered. This paper describes an
approach to identify entire function bodies using calling
conventions, induction analysis, and control flow. In
Section 7 we will present experimental results indicating
that our approach is able to recognize 198% more function
bodies then the use of calling conventions alone.

3 Overview of Compiler

This section provides an overview of our compiler
framework [19][20][21] that automatically translates
software binaries to Register Transfer Level VHDL for
mapping to FPGAs [1]. The compiler’s infrastructure is
shown in Figure 1. The compiler was designed to have a
common entry point for all assembly languages. To this
effort, the front-end requires a description of the processor
ISA in order to configure the assembly language parser. It
uses ISA specifications written in SLED, from the New
Jersey Machine-Code toolkit [4], coupled with a new
semantic description language designed for this project.
The parser generates a virtual assembly representation
called the Machine Language Syntax Tree (MST). This
representation is similar to the MIPS ISA in syntax, but is
generic enough to encapsulate most ISAs, including those
that support predicated and parallel instruction sets. All
MST instructions are three-operand, predicated
instructions. One or more of the operands may be null. The
Control and Data Flow Graph (CDFG) is generated from
the MST, and represents the data dependencies and the
flow of control. Static-single variable assignment (SSA) is
used to break the register name dependencies.

Several traditional optimizations on the CDFG have
been implemented. They serve to reduce the design area
and power consumption, and increase the frequency. The
optimized CDFG is translated into another intermediate
language, analogous to a high-level Hardware Description
Language (HDL). The HDL models processes,
concurrency, and finite state machines. Architecture
specific information is acquired via the Architecture
Description Language (ADL) files. This includes data
pertaining to resource availability, signal names, etc.
Memory models are generated in HDL as required by
backend synthesis tools, such as Synplify Pro [18], to
automatically infer both synchronous and asynchronous
RAMs. The complete HDL is translated directly to RTL
VHDL and Verilog to be mapped onto FPGAs.

4 Motivation

In order to test the automated function extraction
algorithm described in this paper, we have selected the
Texas Instruments C6000 DSP processor architecture [3]
and assembly language as the DSP processor platform, and
Code Composer Studio as the compiler suite. The binary
has been disassembled using the Code Composer Studio
disassembler.
The fragment of TI assembly in Figure 2 can be used to illustrate the extraction problem. Each instruction has been shown along with its address. In the C6000 architecture, the branch instruction executes after five clock cycles. Hence there are 5 delay slots that need to be filled. For example, the “B B3” instruction in address 0x05D4 is followed by five NOP instructions. The TI C6000 ISA provides no special instructions for function calls. The call instructions are syntactically identical to normal branches. This is demonstrated in Figure 2. The code contains three functions ‘main’, ‘CallAddEx’ and ‘add_ex’ and two function calls: ‘main’ calls ‘CallAddEx’ at 0x0610 and ‘CallAddEx’ calls ‘add_ex’ at 0x05B8. The ‘main’ function also has a post-test-loop at lines 0x0610–0x0638. The instruction at 0x0638 is the loop branch. Except for the presence of the predicate at line 0x0638, the branches at 0x0610 and 0x0638 are identical in syntax and behavior. The second function, ‘CallAddEx’, takes a function pointer as an argument in register A4. The branch at 0x05B8 is a call to this pointer. But it is still hard to distinguish this from the other two branches.

To extract function bodies, normal branches need to be distinguished from the calls. For some of the branch instructions, the differentiation is accomplished using the concept of prologues and epilogues [12]. For example, for the TI chip series the caller prologue needs the return address to be moved to register ‘B3’ before the branch is executed. On the other hand, the callee epilogue consists of a jump to register ‘B3’. However, it might not be possible to determine these destinations in all cases. For example, it is not clear by simple inspection if the branch at instruction 0x05B8 is to function ‘add_ex’. This would require knowledge of the input parameter at compile time and may not be available for complicated real world applications. Thus, if the destination of the call to ‘add_ex’ is not recognized, ‘add_ex’ will not be recognized as a function using caller-prologues alone.

A function might have multiple return calls embedded within conditional blocks. It might also have trailing dead code that has not been optimized away. These properties make it difficult to identify function boundaries using callee epilogues. Since symbolic information may not be available, using callee prologues to mark the beginning of the next function is hard as well. These limitations of prologues and epilogues suggest the need for a more complete technique for identifying function bodies.

5 Function Extraction

Our main contribution in this paper is an algorithm to extract function bodies from the binaries, where the function boundaries are not clear. In other words, by inspecting the assembly instructions in a binary, we aim to extract function boundaries. This will later be used to partition the application into hardware and software components.

![Figure 2. Simple code with function calls](image-url)
final list of function bodies. During this process, the heuristic needs to maintain information on the identified functions. It also needs a function mapping instruction addresses to labels and one mapping labels to instruction pointers. This information is maintained as three separate hash structures to reduce processing time.

Finally, a function call graph is generated. This is used to identify procedures that can be moved to hardware. The list of return addresses is used to generate switch-case structures to mimic function returns. Ongoing work on hardware/software partitioning will try to automate the selection process. However, such techniques are out of the scope of this paper. We measure the success of our algorithm by the fraction of functions discovered.

In the following section, we first discuss how the extracted function bodies will be implemented. In other words, we will present the introduction of MST instructions that implement function calls. In the remainder of this section, we will discuss the details of our three-stage algorithm.

5.1 Implementing Control Flow

One important problem we have to address is how to implement the control flow in hardware. The function calls in software are implemented using dynamic jump instructions. However, hardware implementations of functions cannot have dynamic jump instructions. The destination of a transition in a finite state machine for a given input has to be determinable 1. Thus the return instructions have to be replaced by switch case type instructions. The return address is compared with a list of possible locations. If a match is found, a branch is made to the determined location. This obviously assumes that all calls to a function can be correctly identified. Once the control destination is determined, the necessary interface (depending on whether the destination function is implemented in hardware or software) also needs to be generated. Note, incorrect extraction of function bodies may result in suboptimal designs as well as increased complexity of reaching design closure.

The approach of making a comparison with a list of possible locations depends on the location of the caller prologue and not on the number of times the call is made. This is because the return address is peculiar to the address of the call instruction. Thus for a call instruction enclosed within a loop there would be only one return destination.

Assume that the hardware implementation uses the same stack as the software running on the processor. Then it is even possible to implement recursive functions using this approach. Within the binary, there is no implicit scope, i.e. the register values have to be preserved explicitly, if at all.

---

1 In our compilation flow, finite state machines are used to generate the control logic of hardware components. However, other methods for generating the control logic have similar requirements.

---

<table>
<thead>
<tr>
<th>PC</th>
<th>OP</th>
<th>S1</th>
<th>S2</th>
<th>DST</th>
<th>COMMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>5a0</td>
<td>CallAdEx:</td>
<td>:</td>
<td>:</td>
<td>:</td>
<td></td>
</tr>
<tr>
<td>5d4</td>
<td>BEQ(0)</td>
<td>$B3, 0x628, RL1</td>
<td>Return</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5d4</td>
<td>GOTO(0)</td>
<td>CTRL_SINK</td>
<td>:Error check</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5d4</td>
<td>CTRL_SINK:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5d4</td>
<td>NOP(0)0x0</td>
<td>;Control Sink</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

**Figure 3. MST code illustrating function returns**

Since the return address is passed as an argument, the callee can always use the switch-case type return structure to determine the correct return destination, which in this case could be a simple jump to a basic block within the hardware implementation.

Figure 3 demonstrates the return construct for the function ‘CallAddEx’. In this figure, a code segment for the function body generated by the parser is shown. The format of the instructions is MST. The instructions at 0x05D4 represent the return call. Note that, a single assembly instruction (from Figure 2) might be translated into several MST instructions. Therefore, the MST uses ‘Timesteps’ as well as program counters to distinguish between instructions. Because of the five delay slots of the branch instructions, the control should return to the instruction at 0x0628 (please refer to Figure 2 for the complete sequence of instructions). This instruction bears the label ‘RL1’. The new code for ‘CallAddEx’ as shown in Figure 3 compares the destination address in register B3 with 0x628. If B3 is equal to 0x628, the code makes the jump to ‘RL1’. Note that unlike the value inside ‘B3’, ‘RL1’ and 0x628 are statically known locations. Since there is a single call to ‘CallAddEx’, the return is represented by one conditional branch only. The last jump instruction is introduced for verification purposes. If all the calls to the function are correctly identified, then the branch will never be taken.

5.2 Function Extraction Algorithm

As we have previously discussed, we have designed a novel three-stage algorithm to extract function bodies from binaries. Each of these stages are explained in the following sections.

**First Pass: Search for Caller Prologues and Callee Epilogues.** This pass traverses the instruction list from top to bottom while searching for caller prologues. Note that both function calls and other conditional codes (such as loops) use branch instructions possibly with register inputs. To make a categorization of the branch instruction, we need to determine the value of the register whenever the destination of a jump is a register. This is a problematic issue since no structural analysis has been performed at this stage. In many instances the destination address is moved into the register just prior to the call. The need for the jump
to a register value might also be caused by long calls. In these instances the moves will probably not be conditional or predicated and will be found within the current basic block. The ones that cannot be identified in this manner may be identified in the later passes. Hence the search for the destination value and the return address is conducted within the basic block enclosing the call. It starts from the closest label or branch and ends at the current branch.

The purpose of this first pass is to simplify the identification process in the subsequent passes. As we will discuss in the following sections, the subsequent stages of our algorithm are more complicated. Building a complete CDFG on the entire program requires $O(e^2)$ time (discussed in Section 6). The pass generates a list of discernible ‘called’ functions while traversing the instruction list. A function is discernible if the call instruction clearly denotes the first instruction within the function body, e.g. jump to a label or an immediate address. In such cases, the destination of the call can be marked as the beginning of a function. As demonstrated earlier, in some cases, this is not clear. This is most often the case when function pointers are passed as arguments to other functions. Caller prologues are also used to identify the possible return addresses for a given function (refer Figure 2 lines 0x0618 and 0x0620).

In Figure 2 the call to ‘CallAddEx’ at 0x0610 and the return address are easily recognized, as are the call at 0x05B8 and the return address 0x05CC. In the latter, however, the destination in register ‘B4’ is not clear. While identifying prologues, the last calculated value within B3 is used if one is not found in the current block. In pipelined code, it is possible for other branches to break up the caller prologue to prevent the call from being made in some circumstances. For the TI code, the return address in B3 is compared to the projected return address; the branch is identified as a prologue only on a match.

Second Pass: Merging Function Bodies. It is possible for branches to be recognized as function calls erroneously. The second pass aims to eliminate such categorizations. Particularly, this pass looks for return calls within function bodies. When none are found, it merges the adjacent function’s body with the function that was being processed. This leads to the pruning of the function list assuming that the function bodies are not scattered as fragments within the binary (which is correct for almost all binary types available in the literature). Callee epilogues are used to recognize functions returns. The function returns are changed to artificial jumps to a new label, a control sink, attached to the end of the instruction list as discussed in Section 5.1. This aids the interval analysis in the third pass. The generated instruction lists are used by the next pass, which also extracts erroneously merged function bodies.

In the example code from Figure 2, the second pass merges ‘add_ex’ and ‘main’ with ‘CallAddEx’. The first pass found no calls to these functions. Hence, even though callee epilogues are recognized in pass two, there is no need to split the instruction list.

Third Pass: Disjoint Set formation. This part of the heuristic works on the individual function bodies one by one and tries to weed out function bodies not recognized in pass one. Particularly, the third pass traverses the list of function bodies generated by pass two and analyzes the basic blocks in each function body to recognize any possible errors from the first pass. To perform this task, first a control and data flow graph is generated from the instruction list. Information from the first pass is used to generate some of the missing edges. The call instructions are connected to the destinations of their corresponding returns. Note that an edge to the called function is not required here. Therefore, we remove such calls. Otherwise the subsequent blocks would not have predecessors and cause a break in control flow. If the destination of a branch is not known, the branch is changed to a “call” to allow subsequent code to remain connected. These calls can be switched back later. This is a conservative approach and might cause multiple function bodies to be combined. However, it prevents individual bodies from being fragmented.

After this step, we perform induction and interval analysis [2]. Induction analysis attempts to identify the values contained in destination registers. At this point in the algorithm it may not be possible to identify all the reaching definitions correctly. Thus induction is limited to instructions within the current basic block only. Interval analysis is used to generate interval graph sequences. These graphs are used to generate disjoint sets (described below).

The presence of multiple nodes, without ancestors, in the last graph of the sequence indicates the possibility of the existence of other functions. The conversion of returns to jumps to the control sink, connects the functions to a common node, the exit or leaf node. If that node, along with its incident edges, is removed from the graph then the different functions will constitute disjoint sets. The identification of such sets within the graph correctly separates the function bodies.

Thus far, we have assumed that the address of the destination register can be found. However, this might not always be true. The destination blocks of such GOTO’s would be candidates for the ancestor-less blocks with branches incident on interior nodes. But these nodes could finally have a control flow branch linking them to the rest of the function body. In such a case, they would still be a part of the correct disjoint set and be merged with the appropriate body. However, an endless loop could lead to a situation where the preceding blocks cannot be linked to the subsequent ones. Such cases cause the algorithm to return suboptimal solutions, i.e. make incorrect categorizations. Structural analysis can be used to identify such loops and connect the orphaned code artificially.
6 Complexity Analysis

In this section, we present an analysis of the complexity of our algorithm and discuss implementation details. In the first pass (prologue and epilogue identification), the information regarding the identified calls is stored using a hash structure. Each instruction in the list is traversed once. If one assumes that the hash function allows a O(1) access time, the time complexity of this pass is O(n), ‘n’ being the number of instructions in the application.

In the second pass, the identification of epilogues requires a traversal of the function’s instruction body. It might require two traversals of each basic block’s instructions, requiring at most 2m operations, ‘m’ being the size of the instruction list. The sum of the instruction list sizes for all the bodies is equal to the size of the binary, ‘n’. Thus this pass’ time complexity is also O(n).

In the third pass, the generation of the CDFG takes O(n) time. If the number of edges in the CDFG is e, then interval analysis and flow graph construction takes O(e) time[9][10][11]. During the generation of the disjoint sets, the removal of the exit block requires O(n) operations, for the ‘n’ preceding blocks. The use of link lists ensures that the set merge operation takes O(1) time. In this list, each block maintains an indicator to its parent set. The merge operation requires an update of this information, which is an O(n) operation. The disjoint sets are generated while examining each of the graph’s edges. This is an O(e) operation. Thus the time complexity of the entire procedure is O(en). If the number of edges is smaller than the number of blocks, then this can be approximated by O(n²).

7 Experimental Results

To test our algorithm, we have used 8 applications listed in Table 1. These applications contain a commercial TI TMS320C6211 code that implements an echo canceling application as well as 7 other fully linked codes. The echo canceller application consists of precompiled libraries and some C code segments. The disassembled binary for this application has 15323 assembly instructions. The average size of all 8 applications is 8800 instructions. The applications are statically compiled. They included the “stdio” and “stdlib” header files and are linked with the “c6200.lib” run-time library. A detailed inspection of the code revealed that most binaries contained a large number of functions that were not being used in the user code. This also accounted for the large file sizes (8800+ instructions).

Table 1 shows the various components of the extraction results for these binaries. Columns 2 and 3 show the time (in seconds) taken for call graph generation and function extraction, respectively. Columns 4-6 show the number of functions recognized by each stage of our algorithm. Majority of the functions are found in passes 1 and 3. In all instances the number of functions removed in stage two is negligible. Column 9 shows the functions recognized by calling conventions [12] (pass 1 – pass 2). It is also apparent that the majority of the functions were not identifiable using just the procedure calling convention (pass 1). Some function calls identified by pass 1 could not
be assigned instruction bodies and were deleted. This was due to the incomplete nature of the selected code fragments. The number of such functions is presented under the column “Null FNs” in Table 1. The total number of functions found (total FNs) is ‘Pass 3 + Null FNs’. The number of functions found in the smaller binaries was similar since most of them were linked from the included libraries. The rightmost column of Table 1 presents a comparison of our algorithm to function extraction using only procedure calling conventions. For all types of applications, our algorithm is more successful in recognizing function calls. Particularly, for the echo canceller application, our algorithm extracts 2.3 times more functions than can be found using the procedure calling conventions.

Figure 6 shows a portion of the call graph for diffeq. It shows a large number of nodes without predecessors. They represent functions linked but never called. Functions with no attached symbolic information have been assigned names based on their start addresses, e.g. REC_C_3e60.

Figure 7 shows the call graph for the diffeq application rooted at the ‘main’ function. As can be seen, the functions of interest are very few. But unless they are extracted properly their bodies would be merged with the other functions linked into the binary, generating enormous hardware designs with considerable wastage. In fact most of the applications failed to fit on the target FPGA initially. Here ‘fopen’ and its sub-tree are from some linked library. Using this call graph a hardware/software partitioning tool can decide which blocks (e.g. ‘diffeq’) to move to hardware and the interface can be automatically generated.

Table 2 shows the results of the implementations of the separated core function (laplace transform, iir, gcd, etc.) from the 7 smaller benchmarks on a Xilinx Virtex II XCV2V250 FPGA. The first column lists the benchmarks. The second column shows the execution time of the function in clock cycles on a TI C6000 DSP processor using the TI Code Composer Studio instruction level simulator. The third column shows the maximum frequency of operation of the TI C6000 processor. The fourth, fifth, and sixth columns show the results of our compiler mapping functions extracted from DSP assembly onto FPGAs in terms of execution time in cycles on a Xilinx FPGA (measured by ModelSim), area of the FPGA.
Table 2. Results of translating TI C6000 DSP assembly programs to Xilinx Virtex II FPGAs.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>TI C6000 processor</th>
<th>Xilinx Virtex II FPGA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cycles</td>
<td>Freq (MHz)</td>
</tr>
<tr>
<td>laplace</td>
<td>74673</td>
<td>300</td>
</tr>
<tr>
<td>matmul</td>
<td>1799064</td>
<td>300</td>
</tr>
<tr>
<td>iir</td>
<td>22987</td>
<td>300</td>
</tr>
<tr>
<td>fir_cmplx</td>
<td>72856</td>
<td>300</td>
</tr>
<tr>
<td>sobel</td>
<td>127495</td>
<td>300</td>
</tr>
<tr>
<td>gcf</td>
<td>268</td>
<td>300</td>
</tr>
<tr>
<td>diffreq</td>
<td>2318</td>
<td>300</td>
</tr>
</tbody>
</table>

implementation (measured in Look Up Tables by Synpify Pro 7.2), and frequency of the design (measured in MHz estimated by Synpify Pro). We can see that using our techniques, it is possible to obtain performance gains of about 3-20x with an FPGA over the DSP processor with respect to clock cycles. In terms of actual execution times, including the impact of the clock frequencies, the performance gains are about 1.7-5.8x with an FPGA over an embedded processor.

8 Conclusions

Conventional processors (such as DSP processors) are unlikely to meet the performance requirements of next generation embedded applications. Therefore, there is an increasing need to migrate various tasks into application-specific hardware components. However, it is unlikely that the applications will be mapped onto hardware in their entirety. Thus, hardware/software co-design techniques will be highly utilized to achieve this migration. Such algorithms can work in the granularity of instructions, loops, or functions. Working on the granularity of functions is very attractive due to small input size. However, in most cases the source code of the application is not available to the designer. Therefore, extraction of functions from binaries is required. This paper presents an algorithm to recognize function bodies in binaries. Particularly, the algorithm generates a call graph for a binary/assembly to HDL compiler. This call graph can be utilized for the SOC design. Experiments have demonstrated that it is effective in identifying most function calls and function bodies. Experiments have demonstrated that most of the functions were not recognizable by the mere use of calling conventions. Specifically, our algorithm is able to recognize 198% more functions, on average. Hence, we believe that our algorithm can be effectively used by various design tools.

9 References