Static Program Analysis to Detect Hard Coded Addresses and its Application to TI's DSP Processor

Ramakrishnan Venkitaraman Gopal Gupta

Applied Logic, Programming-Languages and Systems (ALPS) Laboratory
Department of Computer Science
The University of Texas at Dallas
Email: *ramakrishnan@student.utdallas.edu, gupta@utdallas.edu*

Abstract

We describe a static analysis based approach to detect hard coded addresses in programs. Our tool takes object code as input, disassembles it, builds the flow-graph, and statically analyzes the flow-graph for the presence of dereferenced pointers that are hard coded. We also elaborate on how our tool can be used to validate DSP software for conformance with TI TMS320 DSP Algorithm Standard's General Programming Rules.

1 Introduction

The TI TMS320 DSP Algorithm Standard defines a set of requirements for DSP algorithms that, if followed, will allow system integrators to quickly assemble production quality systems from one or more algorithms. The standard aims to provide a framework that will enable the development of a rich set of Commercial Off-The-Shelf (COTS) components marketplace for the DSP algorithm technology that will significantly reduce the time to market for new DSP based products. There are 34 rules and 15 guidelines defined by the standard.

Tools from TI are available that check for the compatibility of an algorithm with most of the rules. Rules 1 through 6 in the standard fall under the category of "General Programming rules" and are not currently checked for compliance [3]. These rules provide a complete *Software Management Solution* primarily dealing with simplifying system integration (i.e., the integration of various disparate algorithms utilized by applications) and increasing software reuse.

This document describes an approach for checking an algorithm (given only its binary code) for its compatibility with the "General Programming Rules" defined by the standard. We give details of the tool that we have built to check for compliance of an algorithm with Rule 3 of the standard. Other Rules can be checked in a similar manner.

2 General Programming Rules

The Rules that fall under the category of "General Programming Rules" are the following. (cf. "TI TMS320 DSP Algorithm Standard Rules and Guidelines"[3]):

1. *All algorithms must follow the run time conventions imposed by TI's implementation of the C programming language.*

2. *All algorithms must be reentrant within a preemptive environment including time sliced preemption.*

3. *All algorithm data references must be fully relocatable (subject to alignment requirements). That is, there must be no "hard coded" data memory locations.*

4. *All algorithm code must be fully relocatable. That is, there can be no hard coded program memory locations.*

5. *Algorithms must characterize their ROM-ability; i.e., state whether they are ROM-able or not.*

6. *Algorithms must never directly access any peripheral device. This includes but is not limited to on-chip DMA's, timers, I/O devices, and cache control registers.*

3 Automatic Detection of Hard Coded Memory Addresses

Rule 3 of the standard says that all data references in an implemented algorithm must be fully relocatable. That is, there must be no hard coded data memory locations. The check for compliance should be made given only the object code for the program, since often the proprietary source code is not available. This problem is significant for the following reasons (for a detailed and complete description refer to [3]):

- Enable system integrators to easily migrate between TI DSP's
- Enable host tools to simplify a system integrators tasks, including configuration, performance modeling, standard conformance, and debugging
- Algorithms from multiple vendors can be integrated into a single system.
- Algorithms are framework-agnostic. That is, the same algorithm can be efficiently used in virtually any application or framework.
- Algorithms can be deployed in purely static as well as dynamic run-time environments.
- Algorithms can be distributed in binary form.
- Integration of algorithms does not require recompilations of the client application; reconfiguration and relinking may be required however.

The problem of detecting hard coded references is to check whether a pointer variable that is assigned a constant value is dereferenced or not in the program. Thus, using the 'C' syntax for illustration purposes, given a variable `p` of type `int *`, we want to check if there is an execution path between a statement of the type `p = k`, where `k` is an expression that yields a constant value, and a later statement containing `*p` (that dereferences a pointer). Of course, the dereferencing may take place directly or indirectly, i.e., we might have an intervening statement `int *q = p` followed by a later statement containing `*q`.

Note that if a program only assigns hard codes a pointer variable but never dereferences it, the program is deemed safe. It is only after such a pointer variable is dereferenced during subsequent execution, that the program is deemed unsafe.

The next section gives a static analysis based approach to detect dereferencing of hard coded memory addresses.

4 A Static Analysis based solution

Using hard coded memory addresses is considered a bad programming practice (except in some situations, e.g., while programming device drivers), and we would like to automatically detect such hard coded references in order to certify a program as relocatable and/or reusable. Detecting if a given pointer points to a fixed memory address is undecidable in general (the proof is trivial and omitted). *Static Program Analysis* is a

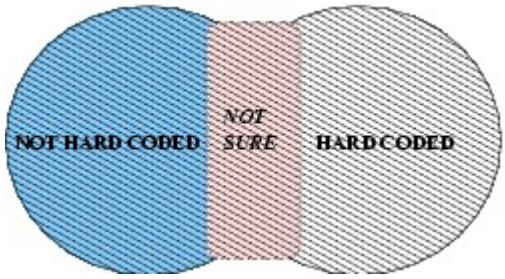

Figure 1: **Interpretations using Static Analysis**

standard tool used to analyze properties that are undecidable. Note, therefore, that, given this undecidability, any static program analysis to detect hard-codedness will only give approximate answers.

Static program analysis (or static analysis for brevity) is defined as any analysis of a program carried out without completely executing the program. Static analysis provides significant benefits and is increasingly recognized as a fundamental tool for analyzing programs. The traditional data-flow analysis found in compiler back-ends is an example of static analysis [5]. Another example of static analysis is *abstract interpretation*, in which a program's data and operations are approximated and the program *abstractly executed* (abstract execution is done to ensure termination) to collect information [6].

Since static analysis can only be approximate, decisive results cannot be obtained. For example, when doing analysis to detect whether the code is hard coded or not, two approaches are possible:

1. The approximation is done in such a way that non hard-coded references may be declared as hard-coded, but hard-coded references will *never* be declared as non hard-coded. Thus, when the analysis result says *"not hard coded"* then the algorithm is certainly compliant but if it says *"hard coded"* then it may or may not be compliant.

2. The approximation is done in such a way that hard-coded references may be declared as non hard-coded, but non hard-coded references will *never* be declared as hard-coded. Thus, when the analysis result says *"hard coded"* then the algorithm is certainly not compliant but if it says its *"not hard coded"* then it may or may not be compliant.

The point to note here is that static analysis based approaches will give the correct solution in most of the cases but may fall into the area of uncertainty for certain kinds of inputs. Note, however, in case of hard-codedness analysis, a programmer would have to go to great lengths to fool the static analyzer; a majority of the cases will be straightforward and easily detectable by the analyzer.

In the following sections, the word *"analyzer"* is used to refer to the tool that we have built to perform static analysis.

4.1 An Overview of Our Approach

The input to the analyzer is the object code (binary) which is to be checked for compliance with the standard. The given object code is disassembled and the corresponding *assembly language code* is obtained. The disassembly is performed using *TI Code Composer Studio* [1, 2]. The disassembled code is provided as input to the analyzer which produces a result which indicates whether the code is compliant with the rule.

In the following discussion when we use the term *"safe"* it means that the program has no hard coded addresses. We use the term *"unsafe"* to mean the opposite.

The algorithm that is implemented by the analyzer has the following steps.

1. Get the disassembled code from the input object code.

2. From the disassembled code, get the basic blocks and construct the flow-graph.

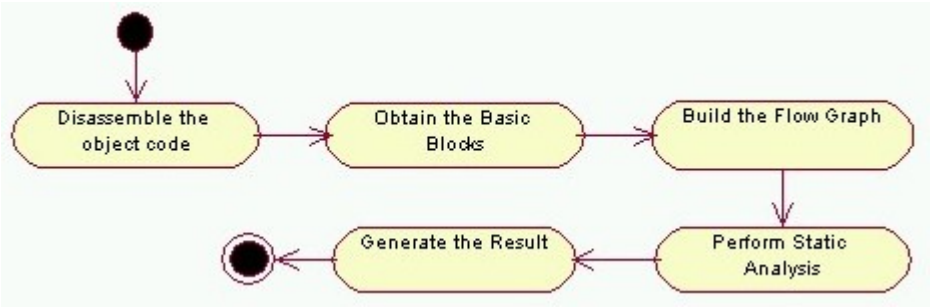

Figure 2: **Activity Diagram for our tool**

3. Analyze the flow-graph and check for the dereferencing of pointer variables.
4. For each such dereferencing, trace back and find out from where did this pointer get its value from (involves the formation of unsafe sets which are explained in the later sections).
5. If the original source of this pointer is hard coded, then declare that the algorithm is not compliant ("unsafe")
6. If the original source from which the pointer got its address value is legitimate then declare that the dereferencing is safe.
7. The algorithm is declared to be safe if and only if all such pointer dereferencing are safe.

Programs can become non-relocatable if data references are hard coded. As explained earlier, data references can be hard coded by allocating say a pointer variable and hard coding the address to which this variable points to. A sample 'C' code that does the same is

```
int *ptr = 0x8800;
```

wherein, the pointer variable *"ptr"* is assigned a *hard coded* address value of 8800 (Hexadecimal). Since the analyzer does not have access to the 'C' code, it checks for the corresponding assembly language statement in the disassembled code.

To check for compliance with the standard, the binary code that is given as input is never executed. The analyzer scans through the disassembled code statically and checks whether there are any hard coded addresses. ***The basic aim of the analysis is to find a path from the point in which the dereferencing occurs to the point at which an address is assigned to the pointer and then check whether that address is legitimate or not.***

One of the first steps in the process is to get the basic blocks from the disassembled code.

4.1.1 Algorithm for Obtaining the Basic Blocks

A *basic block* is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halting or possibility of branching except at the end. The basic blocks are constructed from the disassembled code.[5]

1. The analyzer first determines the set of *leaders*, the first statements of the basic blocks. The rules used are the following.
   (a) The first statement is a leader.
   (b) Any statement that is the target of a Conditional or an Unconditional branch statement ("goto" statement, function calls and like) is a leader.

(c) Any statement that immediately follows a Branch or Conditional Branch statement is a leader

2. For each leader, its basic block consists of the leader and all statements up to but not including the next leader or the end of the program.

The next step is to construct the *control flow-graph* with the basic blocks and is explained in the next section.

4.1.2 Constructing the Control Flow Graph

The basic blocks form the nodes in a directed graph called the *control flow-graph* [5]. The control flow-graph will help us to visualize and arrive at all possible paths through which program control could flow at runtime. All such paths must be analyzed for compliance.

In the control flow-graph, one node is distinguished as the initial node and it is the block whose leader is the first statement. There is a directed edge from node A to B if B can immediately follow A in some execution sequence [5]. That is, if:

1. There is a conditional or unconditional jump from the last statement of A to the first statement of B or

2. B immediately follows A in the order of the program and A does not end in an unconditional jump.

A is said to be the *predecessor* of B and B the *successor* of A.

4.1.3 Analysis of the Flow Graph

The next step is to analyze the flow-graph for compliance with the standard. There are two phases that are involved in the analysis.

Phase 1: *In this phase, the analyzer detects the statements in the disassembled code which correspond to the dereferencing of pointer variables.*

Phase 2: *In this phase, the analyzer checks whether any dereferencing detected in phase 1 is safe*

The above two phases are performed until the analyzer is able to conclude that all the dereferencing that occurs in the program is safe or that the program has a hard coded address (In fact, this detection of dereferencing and check for safety is done for all detected dereferencing operations in parallel).

Detecting Dereferencing (Phase 1): To accomplish this the analyzer starts from the first node in the flow-graph which is a basic block and visits every node in the flow-graph one by one. While visiting any node in the flow-graph it checks for the occurrences of pointer dereferencing. Dereferencing of a pointer is detected in the disassembled code when a register other than the stack pointer(SP) is used as the base register.

Once phase 1 detects the dereferencing of a register say "Reg", a new set called the *Unsafe Set* (which is basically the set of registers which may potentially contain hard coded references) is constructed. Initially, the unsafe set is empty; once phase 1 detects the dereferencing of a register, it adds that register to the set. So if register "Reg" is seen as being dereferenced, it is added into the unsafe set, which will now appear as {Reg}. At this point, the analyzer remembers the line number and block number in which dereferencing occurred (this is the place where it needs to return when phase 2 completes) and *transfers control to phase 2.*

Checking if dereferencing is safe (Phase 2): The check for whether the dereferencing is safe is made by *scanning backward in the control flow-graph.* By "scanning backward" we mean that if the dereferencing of a pointer is detected at line number say with index 'n' then it must have got its value from lines whose index is less than 'n', so the analyzer needs to go up (scan backward) and find the answer to the question "Where did this pointer get its value/address from?"

While scanning backward in phase 2, the analyzer looks for statements in which an element from the unsafe set (in this case 'Reg') is used as the destination register. When it detects such an occurrence say which corresponds to a statement like *"Reg = Reg1 + Reg2"*, it *deletes* the element *"Reg"* from the unsafe set and *inserts* the elements *"Reg1"* and *"Reg2"* into the unsafe set. So now the unsafe set will look like {*Reg1,Reg2*} .

The analyzer continues scanning backward with the current unsafe set (looking out for the occurrence of both Reg1 and Reg2 as the destination registers in this case). The process of scanning backward terminates either when the analyzer concludes that all the elements in the unsafe set are hard coded or when it detects that at least one of the elements in the unsafe set is not hard-coded.

Note that while scanning backward in the disassembled code, the analyzer takes into consideration a subgraph of the control flow-graph that it had already built. This subgraph consists of the nodes in the flow-graph which have a predecessor relation with the node in which the dereferencing occurred. The analyzer performs a Breath First Search and visits every node that is the predecessor of this node (including transitive predecessor relationship); until it is able to conclude if the dereferencing is safe or not. To avoid traversing all paths in the flow-graph (number of execution paths may be exponential in the number of nodes in the flow-graph), the analyzer will keep combining information about a variable along various execution paths (this combining involves performing lattice theoretic operations such as computing *least upper bounds* [7, 6]; however, we omit the details to keep the presentation non-technical). If a loop is encountered during this search, then this loop is traversed (along with merging of information gathered by computing least upper bounds) until the analyzer concludes that no new information is being generated (i.e., a fixed point has been reached). Once a fixed point is reached, search moves to the block that is a predecessor of the lowest numbered block in the loop.

Note that static analysis will stop only when a fixedpoint has been reached for the whole program flow-graph.

The analyzer adopts a set of criteria to decide whether a dereferencing is hard coded or not. As already mentioned, a register corresponding to a pointer is hard coded if its assigned a constant value as in the statement *"Reg=0x8800"*. Some of the ways in which an address could be *legitimate* are:

- If the address is derived from a call to memory allocation routines like "malloc" or "calloc"
- If the address is derived as a function of the "stack pointer"
- If the address is derived from another pointer that is legitimate.

Based on the above the following cases can be distinguished. We illustrate each case with a simple example.

- A *dereferenced pointer* is **NOT Hard Coded** even if one of the elements in the unsafe set is not hard coded.

  Example: Consider the following 'C' code fragment (Note: Some examples throughout this document are given as 'C' code fragments for clarity though the real analysis is done at the assembly language level)

```
int *p, *q, *r, val; //line 0
p = (int*)malloc(sizeof(int)); //line 1
q = (int*)0x8000; // line 2
r = p + q; // line 3
val = *r; // line 4
```

In the above example, the pointer "p" is legitimate and the pointer "q" is hard coded. The pointer "r" is derived from the pointers "p" and "q". The pointer "r" is dereferenced and is assigned to the variable "val". Clearly, *"r" is not a hard coded pointer* as it is derived as a function of a legitimate pointer "p".

When the variable "r" is dereferenced in line 4, the unsafe set will get initialized to {r}. When the assignment statement in line 3 is encountered, the unsafe set gets modified to {p,q}. Analysis of line 2 concludes that "q" is hard coded and analysis of line 1 concludes that the pointer "p" is legitimate. Since "p" is safe and "r" is derived as a function of "p", dereferencing of the pointer "r" in line 4 is also "safe".

- A *dereferenced pointer* is **Hard Coded** if and only if all the elements in the corresponding unsafe set are detected to be hard coded

  Example: Consider the following 'C' code fragment.

```
int *p, *q, *r, val; //line 0
p = (int*) 0x8800; //line 1
q = (int*)0x8000; // line 2
r = p + q; // line 3
val = *r; // line 4
```

  In the example above, the pointers "p" and "q" are hard coded. The pointer "r" is derived from pointers the "p" and "q". The pointer "r" is dereferenced and is assigned to the variable "val". Clearly, *"r" is hard coded* as it is derived from two pointers that were hard coded.

  When the variable "r" is dereferenced in line 4, the unsafe set will be initialized to {r}. When the assignment statement in line 3 is encountered, the unsafe set gets modified to {p,q}. Analysis of line 2 concludes that "q" is hard coded and further analysis reveals that the pointer "p" is also hard coded (line 1). Since all the elements in the unsafe set namely "p" and "q" are hard coded, the original dereferencing of "r", to which the current unsafe set corresponds to, is also declared "hard coded".

- A *program* is said to be **hard coded** even if one of the pointers that's dereferenced is detected to be hard coded.

  Example: Consider the following 'C' code fragment.

```
int *p, *q, v1, v2; //line 0
p = (int*) malloc(sizeof(int)); // line 1
q = (int*) 8000; // line 2
v1 = *p; // line 3
v2 = *q; // line 4
```

  In the example code above there are two pointers out of which, clearly "p" is legitimate and "q" is hard coded. The analyzer first detects the dereferencing of the pointer "q" (line 4) and after analysis concludes that the dereferencing of that pointer is safe (since the address is derived from a call to the memory allocation routine "malloc" in line 1). At this point the analyzer cannot declare that the program is safe and is free of hard coded pointers. The analyzer needs to make sure that other pointer dereferencing that occurs in the program is also safe. The dereferencing of the pointer "q" is unsafe as the pointer "q" gets hard coded in line 2. So, though the assignment in line 3 is safe, *the unsafe assignment in line 4 makes the entire program unsafe.*

- A *program* is **NOT Hard Coded** if and only if all the pointers that are dereferenced are detected to be not hard coded.

Example: Consider the following 'C' code fragment.

```
int *p, *q, v1, v2; //line 0
p = (int*) malloc(sizeof(int)); // line 1
q = (int*) calloc(1, sizeof(int)); // line 2
v1 = *p; // line 3
v2 = *q; // line 4
```

In the example code above, there are two pointer declarations and both of them are derived legitimately (lines 1 and 2). The analyzer will detect the dereferencing of the pointers in lines 3 and 4 respectively. Further analysis of the code will reveal that the pointers are not hard coded (lines 1 and 2). Since there are only two pointers that are dereferenced in the code and since both of them are safe, *the entire code is safe.*

4.2 Handling assignments to pointers from other variables

The only way a pointer variable can get a value is through some type of an assignment statement (the assignment operator, an input read operation, etc.). This assignment may give it a value which is either legitimate or illegitimate. There can also be cases in which pointers are assigned values from integer variables. But just because its assigned a value from an integer variable does not mean that the pointer is hard coded. Consider the following example.

```
int *p, *q, val = 5; //line 0
q = malloc(sizeof(int)); //line 1
val = (int) q; //line 2
p = (int*) val; //line 3
val = *p; //line 4
```

In the example above though the pointer "p" got its value from an integer value in line 3, it is not assigned a hard coded constant because the integer variable (namely "val") got its value from a legitimate pointer in line 2. So the dereferencing in line 4 is safe and hence the entire piece of code is safe.

Our careful formation of unsafe sets and analysis makes sure that if the above code is provided as input to our analyzer, it will give the correct result, that is, *the above program is safe.*

4.3 Handling Function Calls and Loops

When performing static analysis of code, the analyzer may come across function calls. Handling of function calls is simple because its considered similar to a branch statement and the source of the function call is the end of a basic block and the target of the function call is the beginning of a new basic block.

Handling looping structures and recursive function calls is slightly more complex. The reason is because loops may be coded to execute say n times and this value n may not be known at compile-time. Thus, when performing static analysis, the analyzer has no idea about how many times this loop will indeed execute!

If not properly handled, loops may result in the formation of wrong unsafe sets, which will lead to incorrect results. When the analyzer builds the unsafe sets during analysis, if it encounters a loop, it will cycle through the loop and keep on updating the unsafe set. The analyzer will remember the sequence of unsafe sets generated. A fix point is reached when a sequence is repeated. That is, sequence of unsafe sets reaches a *fixed point.*

4.4 Handling Parallelism

The $\|$ characters in the disassembled code signify that an instruction is to execute in parallel with the previous instruction.[4]

```
   instruction A
|| instruction B
|| instruction C
```

For example, if a code sequence as shown is encountered, where instructions A, B, C are some assembly level instructions then it means that instructions A, B and C are executed in parallel. That is, the instructions A, B and C in the fetch packet correspond to the same execute packet and are executed in the same cycle. Moreover, instructions A, B and C do not use any of the same functional units, cross paths, or other data path resources.

Static analysis of the disassembled code needs to make sure that it handles such kind of parallelism. The analyzer does take care of parallelism. As soon as dereferencing of a base register or occurrence of an element in the unsafe set (as the destination register) is found to occur in parallel with other instructions, the analyzer skips the parallel instructions and continues analysis with the instructions that occur in the previous cycle.

5 Current Status

We have now finished building a prototype that tests for hard coding of data memory locations(compliance with Rule 3). In the following discussion all the line and block indices are assumed to start at index 0. Some of the sample input and the output produced by our tool is given below. Only simple examples have been selected, to keep the presentation simple.

Example 1: Sample Disassembled code which is compliant

```
000014C0          main:
000014C0 01BC54F6           STW.D2T2      B3,*SP--[0x2]
000014C4 00002000           NOP           2
000014C8 0FFDDE10           B.S1          malloc
000014CC 01856162           ADDKPC.S2     RL0,B3,3
000014D0 02008040           MVK.D1        4,A4
000014D4          RL0:
000014D4 023C22F4           STW.D2T1      A4,*+SP[0x1]
000014D8 00002000           NOP           2
000014DC 02101AF3           MV.D2X        A4,B4
000014E0 0280052A  ||       MVK.S2        0x000a,B5
000014E4 029002F6           STW.D2T2      B5,*+B4[0x0]
000014E8 00002000           NOP           2
000014EC 01BC52E6           LDW.D2T2      *++SP[0x2],B3
000014F0 00006000           NOP           4
000014F4 008CA362           BNOP.S2       B3,5
000014F8 00000000           NOP
000014FC 00000000           NOP
```

Result produced by our Tool: "The code is compliant with Rule3"

Explanation: The above piece of code consists of only one basic block. During phase 1 of the analysis, the analyzer finds that the base register B4 gets dereferenced(in line 11). So the unsafe set gets initialized to {B4}. Scanning backward (phase 2 in the analysis), it finds that the register 'B4' is assigned a value from the register 'A4'. So the contents of the unsafe set gets modified to {A4}. Scanning backward and looking for 'A4', the analyzer finds that the register 'A4' got its value from a call to the memory allocation routine "malloc". So, 'A4' is a legitimate pointer. So the dereferencing of the register 'B4'(in line 11) is declared to be safe.

Example 2: Sample Disassembled code which is NOT compliant

```
00001440          main:
00001440 01BCD4F6             STW.D2T2      B3,*SP--[0x6]
00001444 00002000             NOP           2
00001448 0FFDEE10             B.S1          malloc
0000144C 01856162             ADDKPC.S2     RL0,B3,3
00001450 02008040             MVK.D1        4,A4
00001454          RL0:
00001454 023C22F4             STW.D2T1      A4,*+SP[0x1]
00001458 00002000             NOP           2
0000145C 020FA02A             MVK.S2        0x1f40,B4
00001460 023C42F6             STW.D2T2      B4,*+SP[0x2]
00001464 00002000             NOP           2
00001468 02002042             MVK.D2        1,B4
0000146C 023C82F6             STW.D2T2      B4,*+SP[0x4]
00001470 00002000             NOP           2
00001474 001008F2             OR.D2         0,B4,B0
00001478 200AA120      [ B0]  BNOP.S1       L1,5
0000147C 000D6120             BNOP.S1       L2,3
00001480 02101AF2             MV.D2X        A4,B4
00001484 023C62F6             STW.D2T2      B4,*+SP[0x3]
00001488          L1:
00001488 023C42E6             LDW.D2T2      *+SP[0x2],B4
0000148C 00006000             NOP           4
00001490 023C62F6             STW.D2T2      B4,*+SP[0x3]
00001494          L2:
00001494 023C62E6             LDW.D2T2      *+SP[0x3],B4
00001498 00006000             NOP           4
0000149C 021002E6             LDW.D2T2      *+B4[0x0],B4
000014A0 00006000             NOP           4
000014A4 023C82F6             STW.D2T2      B4,*+SP[0x4]
000014A8 00002000             NOP           2
000014AC 01BCD2E6             LDW.D2T2      *++SP[0x6],B3
000014B0 00006000             NOP           4
000014B4 008CA362             BNOP.S2       B3,5
000014B8 00000000             NOP
```

Result produced by our tool:

Hard Coding "MVK" detected @ Block Index = 0 and Line Index = 9
0000145C 020FA02A MVK.S2 0x1f40,B4
The code is NOT compliant with Rule3

Explanation: The assembly language code in example 2 has four basic blocks and the corresponding flow-graph is shown in the figure 3. Phase 1 of the analysis detects the dereferencing of the register 'B4' in block 3. The contents of the unsafe set is initialized to {B4} and phase 2 begins. Scanning backward in block 3 the analyzer finds that that 'B4' got its value from the contents of the location 'SP[0x3]'. So the unsafe set gets modified to {*+SP[0x3]}. Since there are no more lines to scan backward in block 3, the predecessors of block 3 namely blocks 1 and 2 are scanned next with the current unsafe set which is {*+SP[0x3]}.

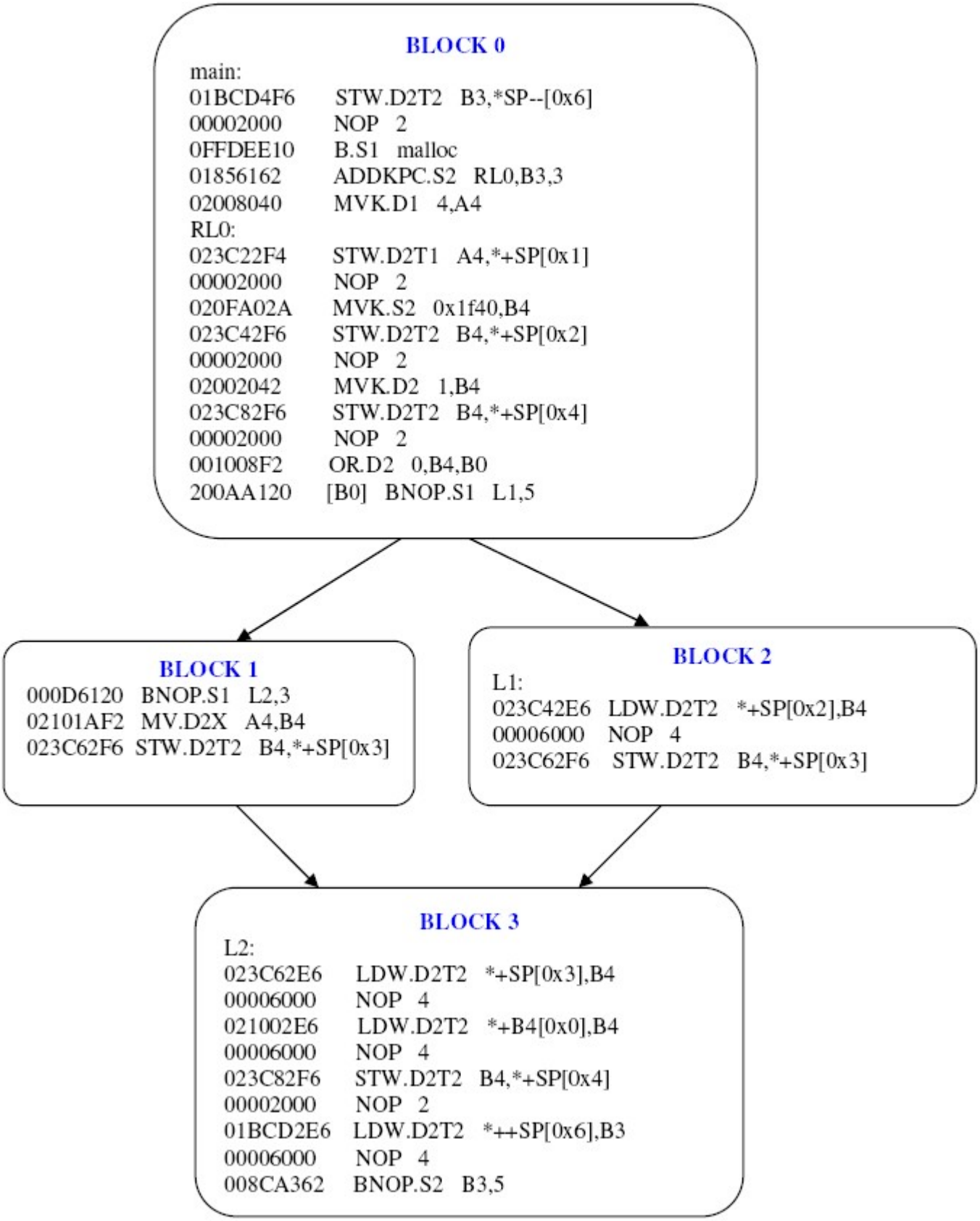

Figure 3: **Flow Graph corresponding to Example 2**

Scanning backward in block 1 with the contents of the unsafe set holding {*+SP[0x3]} detects that '*+SP[0x3]' came from 'B4'. Now the contents of the unsafe set become {B4}. Continuing scanning backward reveals that 'B4' came from 'A4'. So the contents of the unsafe set get modified to {A4}. Since there are no more lines to scan in this block, the predecessor of this block namely block 0 (transitive predecessor of block 3) is scanned with the contents of the unsafe set holding {A4}. Analysis of block 0 concludes that 'A4' is a legitimate pointer as it is derived from a call to "malloc". Since 'A4' is safe, the original dereferencing of the pointer B4 in block 3 *across this path* is safe.

Note that at this point the analyzer cannot declare the dereferencing of base register 'B4' in block 3 to be completely safe because the block in which the dereferencing occurred (block 3) has 2 predecessors (block 1 and 2) and analysis has now concluded the dereferencing to be safe for the path through block 1. The path through block 2 is yet to be checked. The analyzer does that next.

Scanning backward in block 2 with the contents of the unsafe set holding {*+SP[0x3]} (the contents of the unsafe set when the analyzer was at the top of block 3), detects that '*+SP[0x3]' came from 'B4'. So the contents of the unsafe set gets modified to {B4}. Continuing scanning backward in block 2 the analyzer finds that 'B4' got its value from the contents at location 'SP[0x2]'. So now the unsafe set is updated to hold {*+SP[0x2]}. Since the analyzer has reached the top of block 2, it scans the predecessor of block 2 namely block 0 with the current contents of the unsafe set. Analysis of block 0(transitive predecessor of block 3) finds that '*+SP[0x2]' came from 'B4'. So the contents of the unsafe set becomes {B4}. The analyzer continues to scan backward and finds that 'B4' is assigned a constant value of "0x1f40" and hence the analysis concludes that 'B4' (in block 0) is hard coded and hence the original dereferencing of register 'B4' in block 3 is also hard coded.

Since the dereferencing of the base register 'B4' in block 3 is hard coded, analysis concludes that the *entire program is unsafe and is not compliant with Rule 3 of the standard.*

6 Checking For Compliance With Other Rules

Rule 2 can be checked for by making sure that all elements of an algorithm are re-entrant and do not maintain state information in the form of static or global variables during successive invocations. This can also be done by statically analyzing the flow-graph in a similar manner.

Rule 4 is similar to Rule 3 and the check is to make sure that no program memory locations are hard coded. That is, to check for statements which make the program memory like the target of a conditional or un-conditional branch instruction to be hard coded. The analyzer needs to find a path from where a "Branch" is specified to the place where the target address for the branch came from.

7 Related Work and Conclusions

There are other approaches that can be used to check for hard coded addresses. One of the approaches is to develop a system based on *Dynamic Analysis*. This process may involve the creation of a system that takes the input object code, runs it, gets the output and compares the produced output with the desired output. Then the algorithm is relocated and re-run. That is, multiple application binaries are created by locating data in slightly different locations, running it and observing the results. However, such a system cannot guarantee correctness since execution is done for certain inputs, which may not even go through the unsafe execution paths. That is, such a system can never unambiguously tell whether the input algorithm is compliant or not. The clear short comings of such a system is one of the reasons for why we chose static analysis as our approach. 'Static Analysis' can give unambiguous results for a larger set of cases when compared to 'Dynamic Analysis' because of the very nature of the approach.

The development and testing of the tool is currently in progress. Our system is implemented in C[8]. Current work includes fine tuning the handling of loops and extending our system for the remaining rules. Our work so far can be regarded as an attempt to demonstrate the efficacy of static analysis to perform these checks.

References

[1] *Texas Instruments Code Composer Studio Getting Started Guide, Literature No: SPRU509C*

[2] *Texas Instruments TMS320C6000 Code Composer Studio Tutorial, Literature No: SPRU301C*

[3] *TI TMS320 DSP Algorithm Standard Rules and Guidelines, Literature No: SPRU352D*

[4] *TI TMS320C6000 CPU and Instruction Set Reference Guide, Literature No: SPRU189F*

[5] Alfred V.Aho, Ravi Sethi and Jeffrey D.Ullman *"Compilers: Principles, Techniques, and Tools", Addison-Wesley 1988*

[6] S. Abramsky and C. Hankin *"Abstract Interpretation of Declarative Languages", Ellis Horwood, 1987.*

[7] D. Schmidt. *"Denotational Semantics", Allyn and Bacon, 1986.*

[8] Brian W.Kernigham and Dennis M.Ritchie *"The C Programming Language", Second Edition, Prentice Hall 2001.*