Stack Size Analysis for Interrupt Driven Programs

Krishnendu Chatterjee[1], Di Ma[2], Rupak Majumdar[1],
Tian Zhao[3], Thomas A. Henzinger[1], and Jens Palsberg[2]

[1] University of California, Berkeley
{c_krish,rupak,tah}@cs.berkeley.edu
[2] Purdue University
{madi,palsberg}@cs.purdue.edu
[3] University of Wisconsin, Milwaukee
tzhao@cs.uwm.edu

Abstract. We study the problem of determining stack boundedness and the exact maximum stack size for three classes of interrupt driven programs. Interrupt driven programs are used in many real-time applications that require responsive interrupt handling. In order to ensure responsiveness, programmers often enable interrupt processing in the body of lower priority interrupt handlers. In such programs a programming error can allow interrupt handlers to be interrupted in cyclic fashion to lead to an unbounded stack, causing the system to crash. For a restricted class of interrupt driven programs, we show that there is a polynomial time procedure to check stack boundedness, while determining the exact maximum stack size is PSPACE-complete. For a larger class of programs, the two problems are both PSPACE-complete, and for the largest class of programs we consider, the two problems are PSPACE-hard and can be solved in exponential time.

1 Introduction

1.1 Background

Most embedded software runs on resource-constrained processors, often for economical reasons. Once the processor, RAM, etc. have been chosen for an embedded system, the programmers has to fit everything into the available space. For example, on a Z86 processor, the stack exists in the 256 bytes of register space, and it is crucial that the program does not overflow the stack, corrupting other data. Estimating the stack size used by a program is therefore of paramount interest to the correct operation of these systems. An upper bound is necessary to check if the program fits into the available memory. This bound needs, moreover, to be tight: otherwise, precious system resources (e.g., registers) are allocated unnecessarily.

Stack size analysis is particularly challenging for *interrupt-driven software*. Interrupt-driven software is often used in embedded real-time applications that require fast response to external events. Interrupt-driven systems usually have a fixed number of external interrupt sources, and for each interrupt source, a handler that services the interrupt. When an external interrupt occurs, control is transferred automatically to the corresponding handler if interrupt processing is enabled. To maintain fast response, interrupts are enabled most of the time, in particular, higher priority interrupts are enabled in lower priority handlers. Interrupt handling uses stack space: when a handler is called, a return address is placed on the stack, and if the handler itself gets interrupted, then another return address is placed on the stack, and so on. A particularly nasty programming error is to allow the interrupt handlers to interrupt each other indefinitely, leading to an unbounded stack. Moreover, since stack boundedness violations can occur only for particular interrupt sequences, these errors are extremely difficult to replicate and debug, and standard testing is often inadequate.

Algorithms that statically check for stack boundedness and automatically provide precise bounds on the maximum stack size will be important development tools for interrupt-driven systems. In this paper, we provide algorithms for the following two problems for a large class of interrupt driven programs:

Stack boundedness problem. Given an interrupt driven program, the stack boundedness problem asks if the stack size is bounded by a finite constant. Formally, given a program p, the stack boundedness problem returns "yes" if there exists a finite integer k such that on all executions of program p, the stack size never grows beyond k.

Exact maximum stack size problem. Given an interrupt driven program, the exact maximum stack size problem asks what the maximum possible stack size can be on any execution. Formally, given a program p, the exact maximum stack size problem returns an integer k such that for all executions of the program p, the stack size never grows beyond k, and such that there is a possible schedule of interrupts and an execution of program p such that the stack size is k, and returns ∞ if there is an execution where the stack can grow unbounded.

We model interrupt-driven programs in the untyped *interrupt calculus* of Palsberg and Ma [4]. The interrupt calculus contains essential constructs for programming interrupt-driven systems. For example, we have found that the calculus can express the core aspects of seven commercial microcontrollers from Greenhill Manufacturing Ltd. A program in the calculus consists of a main part and some interrupt handlers. In the spirit of such processors as the Intel MCS-51 family (8051, etc.), Motorola Dragonball (68000 family), and Zilog Z86, the interrupt calculus supports an interrupt mask register (imr). An imr value consists of a master bit and one bit for each interrupt source. For example, the Motorola Dragonball processor can handle 22 interrupt sources. For an interrupt handler to be enabled, *both* the master bit and the bit for that interrupt handler have to be set. When an interrupt handler is called, a return address is stored on the stack, and the master bit is automatically turned off. At the time of return, the master bit is turned back on (however, the handler can turn the master bit on at any point). A program execution has access to:

- the interrupt mask register, which can be updated during computation,
- a stack for storing return addresses, and
- a memory of integer variables; output is done via memory-mapped I/O.

Each element on the stack is a return address. When we measure the size of the stack, we simply count the number of elements on the stack.

1.2 Our Results

We consider three versions of Palsberg and Ma's interrupt calculus, here presented in increasing order of generality:

Monotonic programs. These are interrupt calculus programs that satisfy the following monotonicity restriction: when a handler is called with an imr value imr_b, then it returns with an imr value imr_r such that $imr_r \leq imr_b$, where $\leq$ is the logical bitwise implication ordering. In other words, every interrupt that is enabled upon return of a handler must have been enabled when the handler was called (but could have possibly been disabled during the execution of the handler).

Calculus	Problem	Complexity	Reference
Monotonic	Stack boundedness	NLOGSPACE-complete	Theorem 1
	Exact maximum stack size	PSPACE-complete	Theorems 2,4
Monotonic enriched	Stack boundedness	PSPACE-complete	Theorems 3,4
	Exact maximum stack size	PSPACE-complete	Theorems 2,4
Enriched	Stack boundedness	PSPACE-hard, EXPTIME	Theorems 3,6
	Exact maximum stack size	PSPACE-hard, EXPTIME	Theorems 2,6

Table 1. Complexity Results

Monotonic enriched programs. This calculus enriches Palsberg and Ma's calculus with conditionals on the interrupt mask register. The monotonicity restriction from above is retained.

Enriched programs. These are programs in the enriched calculus, without the monotonicity restriction.

We summarize our results in Table 1. We have settled the complexity of stack boundedness and exact maximum stack size both for monotonic programs and for monotonic programs enriched with tests. For general programs enriched with tests, we have a PSPACE lower bound and an EXPTIME upper bound for both problems; tightening this gap remains an open problem.

We reduce the stack boundedness and exact stack size problems to state space exploration problems over certain graphs constructed from the interrupt driven program. We then use the structure of the graph to provide algorithms for the two problems. Our first insight is that for monotonic programs, the maximum stack bounds are attained without any intermediate handler return. The polynomial time algorithm for monotonic programs is reduced to searching for cycles in a graph; the polynomial space algorithm for exact stack bound for monotonic enriched programs is based on the longest path in a (possibly exponential) acyclic graph. Finally, we can reduce the stack boundedness problem and exact maximum stack size problem for enriched programs to finding *context free* cycles and context free longest paths in graphs. Our EXPTIME algorithm for enriched programs is based on a novel technique to find longest context free paths in a DAG. Our lower bounds are obtained by reductions from reachability in a DAG (which NLOGSPACE-complete), satisfiability of quantified boolean formulas (which is PSPACE-complete), and reachability for polynomial-space Turing Machines (which is PSPACE-complete).

We also provide algorithms that determine, given an interrupt driven program, whether it is monotonic. In the nonenriched case, monotonicity can be checked in polynomial time (NLOGSPACE); in the enriched case, in co-NP.

In Section 2, we recall the interrupt calculus of Palsberg and Ma [4]. In Section 3, we consider monotonic programs, in Section 4, we consider monotonic enriched programs, and in Section 5, we consider enriched programs without the monotonicity restriction.

1.3 Related Work

It is possible to cast stack size analysis as a type checking problem. Palsberg and Ma [4] present a type system for the interrupt calculus that guarantees stack boundedness and certifies that the stack size is within a given bound. Each type contains information about the stack size and serves as documentation of the program. However, this requires extensive annotations from the programmer (especially since the types can be exponential in the size of the program), and the type information is absent in legacy programs. Our work can be seen as related to *type inference* for the interrupt calculus. In particular, we

check stack properties of programs without annotations. Thus, our algorithms work for existing, untyped programs. Moreover, from our algorithms, we can infer the types of [4].

Brylow, Damgaard, and Palsberg [1] run a context free reachability algorithm for model checking a carefully chosen *abstraction* of interrupt driven programs. Thus, their approach gives conservative answers to questions about the stack. In contrast to Brylow, Damgaard, and Palsberg [1], we do not use abstraction and therefore we get precise, rather than approximate, results.

Hughes, Pareto, and Sabry [3, 6, 7] use *sized types* to reason about liveness, termination, and space boundedness of reactive systems. However, they require types with explicit space information, and do not address interrupt handling. The complexity of the problem is also not considered.

Wan, Taha, Hudak [9, 10] present two variations of Functional Reactive Programming (FRP) [2] called real-time FRP and event-driven FRP. Both are designed such that the time and space behavior of a program are necessarily bounded. Especially event-driven FRP is related to the interrupt calculus. A main difference between event-driven FRP and the interrupt calculus is that event-driven FRP programs are written in continuation-style, while interrupt calculus programs are written in direct style. So, event-driven FRP programs do not need a stack, and hence stack boundedness is not among the resource issues considered by Wan, Taha, and Hudak.

2 The Interrupt Calculus

2.1 Syntax

We recall the (abstract) syntax of the interrupt calculus of [4]. We use x to range over a set of program variables, we use *imr* to range over bit strings, and we use c to range over integer constants.

$$
\begin{array}{lll}
\text{(program)} & p & ::= (m, \bar{h}) \\
\text{(main)} & m & ::= \mathsf{loop}\ s \mid s\ ;\ m \\
\text{(handler)} & h & ::= \mathsf{iret} \mid s\ ;\ h \\
\text{(statement)} & s & ::= x = e \mid \mathsf{imr} = \mathsf{imr} \wedge imr \mid \mathsf{imr} = \mathsf{imr} \vee imr \mid \\
 & & \quad \mathsf{if0}\ (x)\ s_1\ \mathsf{else}\ s_2 \mid s_1\ ;\ s_2 \mid \mathsf{skip} \\
\text{(expression)} & e & ::= c \mid x \mid x + c \mid x_1 + x_2
\end{array}
$$

The pair $p = (m, \bar{h})$ is an *interrupt program* with main program m and interrupt handlers $\bar{h}$. The over bar notation $\bar{h}$ denotes a sequence $h_1 \ldots h_n$ of handlers. We use the notation $\bar{h}(i) = h_i$. We use a to range over m and h.

2.2 Semantics

We use R to denote a *store*, that is, a partial function mapping program variables to integers. We use σ to denote a *stack* generated by the grammar $\sigma ::= \mathsf{nil} \mid a :: \sigma$. We define the size of a stack as $|\mathsf{nil}| = 0$ and $|a :: \sigma| = 1 + |\sigma|$.

We represent the imr as a bit sequence $imr = b_0 b_1 \ldots b_n$, where $b_i \in \{0, 1\}$. The 0th bit b_0 is the master bit, and for $i > 0$, the ith bit b_i is the bit for interrupts from source i, which are handled by handler i. Notice that the master bit is the most significant bit, the bit for handler 1 is the second-most significant bit, and so on. This layout is different from some processors, but it simplifies the notation used later. For example, the imr value `101b` means that the master bit is set, the bit for handler 1 is not set, and the bit for handler 2 is set. We use the notation $imr(i)$ for bit b_i. The predicate *enabled* is defined as

$$enabled(imr, i) = (imr(0) = 1) \wedge (imr(i) = 1), \qquad i \in 1..n.$$

We use 0 to denote the imr value where all bits are 0. We use t_i to denote the imr value where all bits are 0's except that the ith bit is set to 1. We will use $\wedge$ to denote bitwise logical conjunction, $\vee$ to denote bitwise logical disjunction, $\leq$ to denote bitwise logical implication, and $\neg(\cdot)$ to denote bitwise logical negation. Notice that $\mathit{enabled}(\mathsf{t}_0 \vee \mathsf{t}_i, j)$ is true if $i = j$, and false otherwise. The imr values, ordered by $\leq$, form a lattice with bottom element 0.

A *program state* is a tuple $\langle \bar{h}, R, \mathit{imr}, \sigma, a\rangle$ consisting of interrupt handlers $\bar{h}$, a store R, an interrupt mask register imr, a stack σ of return addresses, and a program counter a. We refer to a as *the current statement*; it models the instruction pointer of a CPU. We use P to range over program states. If $P = \langle \bar{h}, R, \mathit{imr}, \sigma, a\rangle$, then we use the notation $P.\mathit{stk} = \sigma$. For $p = (m, \bar{h})$, the initial program state for executing p is $P_p = \langle \bar{h}, \lambda x.0, 0, \mathsf{nil}, m\rangle$, where the function $\lambda x.0$ is defined on the variables that are used in the program p.

A small-step operational semantics for the language is given by the reflexive, transitive closure of the relation $\rightarrow$ on program states:

$$\langle \bar{h}, R, \mathit{imr}, \sigma, a\rangle \rightarrow \langle \bar{h}, R, \mathit{imr} \wedge \neg \mathsf{t}_0, a :: \sigma, \bar{h}(i)\rangle \quad \text{if } \mathit{enabled}(\mathit{imr}, i) \tag{1}$$

$$\langle \bar{h}, R, \mathit{imr}, \sigma, \mathsf{iret}\rangle \rightarrow \langle \bar{h}, R, \mathit{imr} \vee \mathsf{t}_0, \sigma', a\rangle \quad \text{if } \sigma = a :: \sigma' \tag{2}$$

$$\langle \bar{h}, R, \mathit{imr}, \sigma, \mathsf{loop}\ s\rangle \rightarrow \langle \bar{h}, R, \mathit{imr}, \sigma, s; \mathsf{loop}\ s\rangle \tag{3}$$

$$\langle \bar{h}, R, \mathit{imr}, \sigma, x = e; a\rangle \rightarrow \langle \bar{h}, R\{x \mapsto \mathit{eval}_R(e)\}, \mathit{imr}, \sigma, a\rangle \tag{4}$$

$$\langle \bar{h}, R, \mathit{imr}, \sigma, \mathsf{imr} = \mathsf{imr} \wedge \mathit{imr}'; a\rangle \rightarrow \langle \bar{h}, R, \mathit{imr} \wedge \mathit{imr}', \sigma, a\rangle \tag{5}$$

$$\langle \bar{h}, R, \mathit{imr}, \sigma, \mathsf{imr} = \mathsf{imr} \vee \mathit{imr}'; a\rangle \rightarrow \langle \bar{h}, R, \mathit{imr} \vee \mathit{imr}', \sigma, a\rangle \tag{6}$$

$$\langle \bar{h}, R, \mathit{imr}, \sigma, (\mathsf{if0}\ (x)\ s_1\ \mathsf{else}\ s_2); a\rangle \rightarrow \langle \bar{h}, R, \mathit{imr}, \sigma, s_1; a\rangle \quad \text{if } R(x) = 0 \tag{7}$$

$$\langle \bar{h}, R, \mathit{imr}, \sigma, (\mathsf{if0}\ (x)\ s_1\ \mathsf{else}\ s_2); a\rangle \rightarrow \langle \bar{h}, R, \mathit{imr}, \sigma, s_2; a\rangle \quad \text{if } R(x) \neq 0 \tag{8}$$

$$\langle \bar{h}, R, \mathit{imr}, \sigma, \mathsf{skip}; a\rangle \rightarrow \langle \bar{h}, R, \mathit{imr}, \sigma, a\rangle \tag{9}$$

where the function $\mathit{eval}_R(e)$ is defined as:

$$\begin{array}{ll} \mathit{eval}_R(c) = c & \mathit{eval}_R(x + c) = R(x) + c \\ \mathit{eval}_R(x) = R(x) & \mathit{eval}_R(x_1 + x_2) = R(x_1) + R(x_2). \end{array}$$

Rule (1) models that if an interrupt is enabled, then it may occur. The rule says that if $\mathit{enabled}(\mathit{imr}, i)$, then it is a possible transition to push the current statement on the stack, make $\bar{h}(i)$ the current statement, and turn off the master bit in the imr. Notice that we make no assumptions about the interrupt arrivals; any enabled interrupt can occur at any time, and conversely, no interrupt has to occur. Rule (2) models interrupt return. The rule says that to return from an interrupt, remove the top element of the stack, make the removed top element the current statement, and turn on the master bit. Rule (3) is an unfolding rule for loops, and Rules (4)–(9) are standard rules for statements. Let $\rightarrow^*$ denote the reflexive transitive closure of $\rightarrow$.

Consider a program execution γ of the form $P_p \rightarrow^* P_i \rightarrow P_{i+1} \rightarrow^* P_j \rightarrow P_{j+1} \ldots$ with $P_i = \langle \bar{h}, R, \mathit{imr}_b, \sigma, a\rangle$ and $P_j = \langle \bar{h}, R', \mathit{imr}', \sigma', a'\rangle$. The handler $\bar{h}(i)$ is *called in* γ *with* imr_b at state P_i and *returns with* imr_r at state P_j if

$$\begin{array}{l} P_i \rightarrow P_{i+1} = \langle \bar{h}, R, \mathit{imr}_b \wedge \neg \mathsf{t}_0, a :: \sigma, \bar{h}(i)\rangle \quad \text{and } \mathit{enabled}(\mathit{imr}_b, i), \\ P_j \rightarrow P_{j+1} = \langle \bar{h}, R', \mathit{imr}_r, \sigma, a\rangle \quad \text{and } \sigma' = a :: \sigma, \end{array}$$

and $P_k.\mathit{stk} \neq \sigma$ for all $i < k \leq j$. We say that there is no handler call in γ between P_i and P_j if for all $i \leq k < j$, the transition $P_k \rightarrow P_{k+1}$ is not a transition of the form (1). Similarly, given an execution $P_p \rightarrow^* P_i \rightarrow^* P_j \cdots$, there is no handler return between P_i and P_j if for all $i \leq k < j$, the transition $P_k \rightarrow P_{k+1}$ is not a transition of the form (2).

```
imr = imr or 111b
loop {
   imr = imr or 111b
}
```

```
handler 1 {
   imr = imr and 101b
   imr = imr or  100b
   iret
}
handler 2 {
   imr = imr and 110b
   imr = imr or  010b
   imr = imr or  100b
   imr = imr and 101b
   iret
}
```

Fig. 1. A program in the interrupt calculus

2.3 Stack Size Analysis

We formally define the problems of stack size analysis that we consider in this paper.

Stack boundedness problem Given an interrupt program p, the stack boundedness problem returns "yes" if there exists a finite integer K such that for all P', if $P_p \rightarrow^* P'$ then $|P'.stk| \leq K$.

Exact maximum stack size problem For a program state P we define $maxStackSize(P)$ as the least $K \geq 0$ such that for all P', if $P \rightarrow^* P'$, then $|P'.stk| \leq K$, and "infinite" in case no such K exists. The exact maximum stack size problem given a interrupt program p returns $maxStackSize(P_p)$.

Figure 1 shows an example of a program in the interrupt calculus. The bit sequences such as `111b` are imr values. Notice that each of the two handlers can be called from different program points where the imr's are different. The bodies of the two handlers manipulate the imr, and both are at some point during the execution open to the possibility of being interrupted by the other handler. However, the maximum stack size is 3. This stack size happens if handler 1 is called first, then handler 2, and then handler 1 again, at which time there are three return addresses on the stack.

3 Monotonic Interrupt Programs

We first define monotonic interrupt programs and then analyze the stack boundedness and exact maximum stack size problems for such programs.

A handler h_i of program p is *monotonic* if for every execution γ of p, if h_i is called in γ with an imr value imr_b and returns with an imr value imr_r, then $imr_r \leq imr_b$. The program p is *monotonic* if all handlers $h_1 \ldots h_n$ of p are monotonic. The handler h_i of p is *monotonic in isolation* if for every execution γ of p, if h_i is called in γ with an imr value imr_b from a state P_i and returns with an imr value imr_r from a state P_j such that there is no handler call between P_i and P_j, then $imr_r \leq imr_b$.

We first show that a program $p = (m, \bar{h})$ is monotonic iff every handler $h_i \in \bar{h}$ is monotonic in isolation. Moreover, a handler h_i is monotonic in isolation iff, whenever h_i is called with imr value $\mathsf{t}_0 \vee \mathsf{t}_i$ from state P_i and returns with imr_r from state P_j, with no handler calls between P_i and P_j, then $imr_r \leq \mathsf{t}_0 \vee \mathsf{t}_i$. These observations can be used to efficiently check if an interrupt program is monotonic: for each handler, we check that the return value imr_r of the imr when called with $\mathsf{t}_0 \vee \mathsf{t}_i$ satisfies $imr_r \leq \mathsf{t}_0 \vee \mathsf{t}_i$.

Proposition 1. *It can be checked in linear time (NLOGSPACE) if an interrupt program is monotonic.*

3.1 Stack Boundedness

We now analyze the complexity of stack boundedness of monotonic programs. Our main insight is that the maximum stack size is achieved without any intermediate handler returns. First observe that if handler h is enabled when the imr is imr_1, then it is enabled for all imr $imr_2 \geq imr_1$. We argue the case where the maximum stack size is finite, the same argument can be formalized in case the maximum stack size is infinite. Fix an execution sequence that achieves the maximum stack size. Let h be the last handler that returned in this sequence (if there is no such h then we are done). Suppose h was called with imr_b from h', and returned with imr_r such that $imr_r \leq imr_b$. If the call to h is omitted then the value of the imr at the point of return from h would have been greater (logical implication ordering) if h was not called. Then repeating the same sequence of statements and same sequence of calls to handlers with h omitted gives the same stack size. Following a similar argument, we can show that all handlers that return intermediately can be omitted without changing the maximum stack size attained.

Lemma 1. *For a monotonic program p, let $P_{\max}$ be a program state such that $P_p \rightarrow^* P_{\max}$ and for any state P', if $P_p \rightarrow^* P'$ then $|P_{\max}.stk| \geq |P'.stk|$. Then there is a program state P'' such that $P_p \rightarrow^* P''$, $|P''.stk| = |P_{\max}.stk|$, and there is no handler return between P_p and P''.*

We now give a polynomial time algorithm for the stack boundedness problem for monotonic programs. The algorithm reduces the stack boundedness question to the presence of cycles in the *enabled graph* of a program. Let $h_1, h_2, \ldots, h_n$ be the n handlers of the program. Given the code of the handlers we build the enabled graph $G = \langle V, E \rangle$ as follows:

- There is a node for each handler, i.e., $V = \{h_1, h_2, \ldots, h_n\}$.
- Let the instructions of h_i be $C_i = i_1, i_2, \ldots, i_m$. There is an edge between (h_i, h_j) if any of the following condition holds
  1. There is l, k such that $1 \leq l \leq m$ and $1 \leq k \leq m$ and $l \leq k$ and the instruction at i_l is $\mathsf{imr} = \mathsf{imr} \vee imr$ with $t_0 \leq imr$, the instruction at i_k is $\mathsf{imr} = \mathsf{imr} \vee imr$ with $t_j \leq imr$ and for all statement i_m between i_l and i_k if i_m is $\mathsf{imr} = \mathsf{imr} \wedge imr$ then $t_0 \leq imr$.
  2. There is l, k such that $1 \leq l \leq m$ and $1 \leq k \leq m$ and $l \leq k$ and the instruction at i_l is $\mathsf{imr} = \mathsf{imr} \vee imr$ with $t_j \leq imr$, the instruction at i_k is $\mathsf{imr} = \mathsf{imr} \vee imr$ with $t_0 \leq imr$ and for all statement i_m between i_l and i_k if i_m is $\mathsf{imr} = \mathsf{imr} \wedge imr$ then $t_j \leq imr$.
     In other words there is an edge $(h_i, h_j) \in E$ if the handler h_i enables the interrupt j. Similarly, there is an edge $(h_i, h_i) \in E$ if the above two conditions are satisfied or the following condition is satisfied
  3. There is l such that $1 \leq l \leq m$ and the instruction at i_l is $\mathsf{imr} = \mathsf{imr} \vee imr$ with $t_0 \leq imr$ and for all statement i_m between i_1 and i_l if i_m is $\mathsf{imr} = \mathsf{imr} \wedge imr$ then $t_i \leq imr$.

Since we do not model the program variables, we can analyze the code of h_i and detect all outgoing edges (h_i, h_j) in time linear in the length of h_i. We only need to check that there is an $\vee$ statement with an imr constant with jth bit 1 and then the master bit is turned on with no intermediate disabling of the jth bit or vice versa. Hence the enabled graph for program p can be constructed in time $n^2 \times |p|$ (where $|p|$ denotes the length of p).

Let G_p be the enabled graph for a monotonic interrupt program p. If G_p has a cycle, then the stack is unbounded, that is, for all positive integer K there is a program state P' such that $P_p \rightarrow^* P'$ and $|P'.stk| > K$. Since cycles in the enabled graph can be found in NLOGSPACE, the stack boundedness problem for monotonic programs is in NLOGSPACE; note that the enabled graph of a program can be generated on the fly in logarithmic space. Hardness for NLOGSPACE follows from the hardness of DAG reachability.

Theorem 1. *Stack boundedness for monotonic interrupt programs can be checked in time linear in the size of the program and quadratic in the number of handlers. The complexity of stack boundedness for monotonic interrupt programs is NLOGSPACE-complete.*

In case the stack is bounded, we can get a simple upper bound on the stack size as follows. Let G_p be the enabled graph for a monotonic interrupt program p. If G_p is a DAG, and the node h_i of G_p has order k in topological sorting order, then we can prove by induction that the corresponding handler h_i of p can occur at most $2^{(k-1)}$ times in the stack. Hence for any program state P' such that $P_p \rightarrow^* P'$, we have $|P'.stk| \leq 2^n - 1$. In fact, this bound is tight: there is a program with n handlers that achieves a maximum stack size of $2^n - 1$. We show that starting with an imr value of all 1's one can achieve the maximum stack length of $2^n - 1$ and the stack remain bounded. We give an inductive strategy to achieve this. With one handler which does not turn itself on we can have a stack length 1 starting with imr value 11. By induction hypothesis, using $n-1$ handlers starting with imr value all 1's we can achieve a stack length of $2^{n-1} - 1$. Now we add the nth handler and modify the previous $n-1$ handlers such that they do not change the bit for the nth handler. The n-th handler turns on every bit except itself, and then turns on the master bit. The following sequence achieves a stack size of $2^n - 1$. First, the first $n-1$ handlers achieve a stack size of $2^{n-1} - 1$ using the inductive strategy. After this, the nth handler is called. It enables the $n-1$ handlers but disables itself. Hence the sequence of stack of $2^{n-1} - 1$ can be repeated twice and the n the handler can occur once in the stack in between. The total length of stack is thus $1 + (2^{n-1} - 1) + (2^{n-1} - 1) = 2^n - 1$. Since none of the other handlers can turn the nth handler on, the stack size is in fact bounded. However, the exact maximum stack size problem can be solved only in PSPACE. We defer the algorithm to the next section, where we solve the problem for a more general class of programs.

3.2 Maximum Stack Size

We now prove that the exact maximum stack size problem is PSPACE-hard. We define a subclass of monotonic interrupt calculus which we call simple interrupt calculus and show the exact maximum stack size problem is already PSPACE-hard for this class.

For imr', imr'' where $imr'(0) = 0$ and $imr''(0) = 0$, define $\mathcal{H}(imr';\ imr'')$ to be the interrupt handler

$$\begin{aligned}&\mathsf{imr} = \mathsf{imr} \wedge \neg imr';\\ &\mathsf{imr} = \mathsf{imr} \vee (t_0 \vee imr'');\\ &\mathsf{imr} = \mathsf{imr} \wedge \neg(t_0 \vee imr'');\\ &\mathsf{iret}.\end{aligned}$$

A program $p = (m, \bar{h})$ is *simple* if the main program h has the form

$$\begin{aligned}&\mathsf{imr} = \mathsf{imr} \vee (imr_S \vee t_0);\\ &\mathsf{loop\ skip}\end{aligned}$$

where $imr_S(0) = 0$, and every interrupt handler in $\bar{h}$ has the form $\mathcal{H}(imr';\ imr'')$. Intuitively, a handler of a simple interrupt calculus program first disables some handlers, then enables other handlers and enables interrupt handling. This opens the door to the handler being interrupted by other handlers. After that, it disables interrupt handling, and makes sure that the handlers that are enabled on exit are a subset of those that were enabled on entry to the handler.

For a handler $h(i)$ of the form $\mathcal{H}(imr';\ imr'')$, we define function $f_i(imr) = imr \wedge (\neg imr') \vee imr''$. Given a simple interrupt calculus program p, we define a directed graph $G(p) = (V, E)$ such that

- $V = \{\ imr\ \mid\ imr(0) = 0\ \}$,
- $E = \{\ (imr, f_i(imr), i)\ \mid\ \mathsf{t}_i \leq imr\}$ is a set of labeled edges from imr to $f_i(imr)$ with label $i \in \{1..n\}$.

The edge $(imr, f_i(imr), i)$ in $G(p)$ represents the call to the interrupt handler $h(i)$ when imr value is imr. We define imr_S as the start node of $G(p)$ and we define $\mathcal{M}(imr)$ as the longest path in $G(p)$ from node imr. The notation $\mathcal{M}(imr)$ is ambiguous because it leaves the graph unspecified; however, in all cases below, the graph in question can be inferred from the context.

Lemma 2. *For a simple program p, we have* $maxStackSize(P_p) = |\mathcal{M}(imr_S)|$.

We now show PSPACE-hardness for simple interrupt calculus. Our proof is based on a polynomial time reduction from the *quantified boolean satisfiability* (QSAT) problem [5]. We first illustrate our reduction by a small example. Given a QSAT instance

$$S = \exists x_2 \forall x_1\ \phi$$
$$\phi = \bigwedge_{i=1}^{2} \bigvee_{j=1}^{2} \phi_{ij} = (x_2 \vee \neg x_1) \wedge (x_2 \vee x_1),$$

we construct a simple interrupt program $p = (m, \bar{h})$. We let $\bar{h} = \{h(i)\ \mid\ i \in \{1..12\}\ \}$. Let $q_i^a, w_i^a, d_{ij},\ \forall i,j \in \{1,2\}, a \in \{0,1\}$ be distinct indices of value ranging from 1 to 12. We use four handlers $h(q_2^0), h(q_2^1), h(w_2^0)$, and $h(w_2^1)$ to encode $\exists x_2$; and use $h(q_1^0), h(q_1^1), h(w_1^0), h(w_1^1)$ to encode $\forall x_1$. Intuitively, $\forall k \in \{1,2\}$, the invocation of the handler $h(q_k^0)$ corresponds to $x_k = true$; and the invocation of the handler $h(q_k^1)$ corresponds to $x_k = false$. We also use the handler $h(d_{ij})$ to encode $\phi_{ij}, \forall i,j \in \{1,2\}$. We want to show that S is satisfiable iff $|\mathcal{M}(t_{q_2^0} \vee t_{q_2^1})| = 10$.

We define $\bar{h}$ such that handler $h(q_2^a)$ enables handlers $h(q_1^0), h(q_1^1)$ and $h(w_2^a), \forall a \in \{0,1\}$ by setting the imr bits corresponding to these handlers to 1. The handler $h(q_1^a)$ enables handlers $h(d_{ij}), h(w_1^a)$ for $a \in \{0,1\}, i,j \in \{1,2\}$. We also define $h(d_{ij})$ for $i,j \in \{1,2\}$ such that $h(d_{ij})$ disables handlers $h(d_{i1}), h(d_{i2})$ by setting the imr bits corresponding to these handlers to 0. Similarly, we let $h(q_2^0)$ and $h(q_2^1)$ disable each other. Also, $\forall a \in \{0,1\}$, $h(w_2^a)$ disables $h(d_{ij}), h(q_1^a), h(w_1^a)$ and $h(w_1^a)$ disables $h(d_{ij})$, $\forall i,j \in \{1,2\}$. If $\phi_{ij} = x_k$, then $h(d_{ij})$ disables $h(w_k^1)$ and If $\phi_{ij} = \neg x_k$, then $h(d_{ij})$ disables $h(w_k^0)$. To avoid cyclic path in $G(p)$, we let $h(i)$ disable itself for each $i \in \{1..12\}$.

Since $h(q_2^0), h(q_2^1)$ disable each other, we have that $|\mathcal{M}(t_{q_2^0} \vee t_{q_2^1})| = 10$ iff $\exists a \in \{0,1\}$ such that $|\mathcal{M}(t_{q_2^a})| = 10$. Since $h(q_2^a)$ enables $h(q_1^0), h(q_1^1), h(w_2^a)$, we have that $|\mathcal{M}(t_{q_2^a})| = 10$ iff $|\mathcal{M}(t_{q_1^0} \vee t_{q_1^1} \vee t_{w_2^a})| = 9$. Since $h(w_2^a)$ disables $h(q_1^a), h(w_1^a), h(d_{ij}), \forall i,j \in \{1,2\}, a \in \{0,1\}$, the edge labeled w_2^a must be the last edge in $\mathcal{M}(t_{q_1^0} \vee t_{q_1^1} \vee t_{w_2^a})$ if its length is 9. Thus, $|\mathcal{M}(t_{q_2^a})| = 10$ iff $|\mathcal{M}'(t_{q_1^0})| = |\mathcal{M}'(t_{q_1^1})| = 4$, where $\mathcal{M}'(imr)$ is the longest path in $G'(p)$ and $G'(p)$ is the subgraph of $G(p)$ with edges labeled d_{ij} removed if $h(d_{ij})$ disables $h(w_2^a)$. Also since $h(q_1^0)$ enables $h(d_{ij}), h(w_1^0)$ and $h(w_1^0)$ disables $h(d_{ij})$ for all $i,j \in \{1,2\}$, we have that $|\mathcal{M}'(t_{q_1^0})| = 4$ iff $|\mathcal{M}''(\bigvee_{i,j\in\{1,2\}} t_{d_{ij}})| = 2$, where $\mathcal{M}''(imr)$ is the longest path in $G''(p)$ and $G''(p)$ is the subgraph of $G'(p)$ with edges labeled d_{ij} removed if $h(d_{ij})$ disables $h(w_1^0)$. Similarly, $|\mathcal{M}'(t_{q_1^1})| = 4$ iff $|\mathcal{M}'''(\bigvee_{i,j\in\{1,2\}} t_{d_{ij}})| = 2$, where $\mathcal{M}'''(imr)$ is the longest path in $G'''(p)$ and $G'''(p)$ is the subgraph of $G'(p)$ with edges labeled d_{ij} removed if $h(d_{ij})$ disables $h(w_1^1)$.

Suppose that $a = 0$, then $|\mathcal{M}''(\bigvee_{i,j\in\{1,2\}} t_{d_{ij}})|$ may only contain edges with label d_{ij} where $h(d_{ij})$ does not disable $h(w_2^0)$ or $h(w_1^0)$, which means that $\phi_{ij} = x_2$ or x_1.

Since $h(d_{ij})$ disables $h(d_{i1}), h(d_{i2})$, $\mathcal{M}''(\bigvee_{i,j\in\{1,2\}} t_{d_{ij}})$ contains at most one edge with label d_{ij} for each $i \in \{1,2\}$. Thus, $|\mathcal{M}''(\bigvee_{i,j\in\{1,2\}} t_{d_{ij}})| = 2$ iff $x_2 = true, x_1 = true$ is a satisfiable truth assignment to ϕ. Similarly, $\mathcal{M}'''(\bigvee_{i,j\in\{1,2\}} t_{d_{ij}})| = 2$ iff $x_2 = true, x_1 =$

$false$ is a satisfiable truth assignment to ϕ. Thus, $|\mathcal{M}(t_{q_2^0})| = 10$ iff given $x_2 = true$, $\forall x_1\ \phi$ is satisfiable. Similarly, $|\mathcal{M}(t_{q_2^1})| = 10$ iff given $x_2 = false$, $\forall x_1\ \phi$ is satisfiable.

Thus, $|\mathcal{M}(t_{q_2^0} \vee t_{q_2^1})| = 10$ iff $\exists x_2, \forall x_1\ \phi$ is satisfiable.

For our example, S is satisfiable since if $x_2 = true$ then for all x_1, ϕ is satisfiable. Correspondingly, we have that $|\mathcal{M}(t_{q_2^0} \vee t_{q_2^0})| = |\mathcal{M}(t_{q_2^0})| = 10$, $|\mathcal{M}'(t_{q_1^0})| = |\mathcal{M}'(t_{q_1^1})| = 4$, and $|\mathcal{M}''(\bigvee_{i,j\in\{1,2\}} t_{d_{ij}})| = |\mathcal{M}'''(\bigvee_{i,j\in\{1,2\}} t_{d_{ij}})| = 2$.

Theorem 2. *The exact maximum stack size problem for monotonic interrupt programs is PSPACE-hard.*

The proof is technical, and given in Appendix A.

4 Monotonic Enriched Interrupt Programs

We now introduce an enriched version of the interrupt calculus, where we allow conditionals on the interrupt mask register. The conditional can test if some bit of the imr is on, and then bitwise-or the imr with a constant bit sequence; or it can test if some bit of the imr is off, in which case imr can be bitwise and-ed with a constant bit sequence. The syntax for *enriched interrupt programs* is given by the syntax from Section 2 together with the following clauses:

(statement) $s ::= \cdots \mid \mathsf{if(test}\ i\text{-}\mathsf{th\ bit\ on)imr} = \mathsf{imr} \vee imr \mid \mathsf{if(test}\ i\text{-}\mathsf{th\ bit\ off)imr} = \mathsf{imr} \wedge imr$

The small-step operational semantics for the two commands above are given below:

$\langle \bar{h}, R, imr, \sigma, \mathsf{if(test}\ i\text{-}\mathsf{th\ bit\ on)imr} = \mathsf{imr} \vee imr'; a\rangle \to \langle \bar{h}, R, imr \vee imr', \sigma, a\rangle$ if $imr(i) = 1$

$\langle \bar{h}, R, imr, \sigma, \mathsf{if(test}\ i\text{-}\mathsf{th\ bit\ on)imr} = \mathsf{imr} \vee imr'; a\rangle \to \langle \bar{h}, R, imr, \sigma, a\rangle$ if $imr(i) = 0$

$\langle \bar{h}, R, imr, \sigma, \mathsf{if(test}\ i\text{-}\mathsf{th\ bit\ off)imr} = \mathsf{imr} \wedge imr'; a\rangle \to \langle \bar{h}, R, imr \wedge imr', \sigma, a\rangle$ if $imr(i) = 0$

$\langle \bar{h}, R, imr, \sigma, \mathsf{if(test}\ i\text{-}\mathsf{th\ bit\ off)imr} = \mathsf{imr} \vee imr'; a\rangle \to \langle \bar{h}, R, imr, \sigma, a\rangle$ if $imr(i) = 1$

Proposition 2. *It can be checked in time exponential in the number of handlers (in co-NP) if an enriched interrupt program is monotonic.*

For monotonic enriched interrupt programs, both the stack boundedness problem and the exact maximum stack size problem are PSPACE-complete. To show this, we first show that the stack boundedness problem is PSPACE-hard by a generic reduction from polynomial space Turing machines. We fix a PSPACE-complete Turing machine M. Given input x, we construct in polynomial time a program p such that M accepts x iff p has an unbounded stack. We have two handlers for each tape cell (one representing zero, and the other representing one), and a handler for each triple (i, q, b) of head position i, control state q, and bit b. The handlers encode the working of the Turing machine in a standard way. The main program sets the bits corresponding to the initial state of the Turing machine, with x written on the tape. Finally, we have an extra handler that enables itself (and so can cause an unbounded stack) that is set only when the machine reaches an accepting state. The details of the reduction can be found in Appendix B.

Theorem 3. *The stack boundedness problem for monotonic enriched interrupt programs is PSPACE-hard.*

We now give a PSPACE algorithm to check the exact maximum stack size. Since we restrict our programs to be monotonic it follows from Lemma 1 that the maximum length of the stack can be achieved with no handler returning in between. Given a program p of m statements and n handlers we label the statements as $pc_1, pc_2, \ldots, pc_m$. Let PC denote the set of all statements i.e $PC = \{pc_1, pc_2, \ldots, pc_m\}$. Consider the graph G_p where there is a node v for every statement with all possible imr values (i.e $v = \langle pc, imr\rangle$ for some value among PC and some imr value). Let $v = \langle pc, imr\rangle$ and $v' = \langle pc', imr'\rangle$ be two nodes in the graph. There is an edge between v, v' in G if any of the following two conditions hold:

– on executing the statement at pc with imr value imr the control goes to pc' and the value of imr is imr'. The weight of this edge is 0.
– pc' is a starting address of a handler , imr has the bit for the handler and the master bit enabled and imr' is same as the value of imr with master bit off. The weight of this edge is 1.

We also have a special node in the graph called *target* and add edges to *target* of weight 0 from all those nodes which correspond to a $pc \in PC$ which is a *iret* statement. This graph is exponential in the size of the input as there are $O(|PC| \times 2^n)$ nodes in the graph. The starting node of the graph is the node with pc_1 and $imr = 000\cdots000$. If there is a node in the graph which is the starting address of a handler h and which is reachable from the start node and also self-reachable then the stack length would be infinite. This is because the sequence of calls from the starting statement to the handler h is first executed and then the cycle of handler calls is repeated infinitely many times. As the handler h is in stack when it is called again the stack would grow infinite. Since there is a sequence of interrupts which achieves the maximum stack length without any handler returning in between (follows from Lemma 1) if there is no cycle in G_p we need to find the longest path in the DAG G_p.

Theorem 4. *The exact maximum stack size for monotonic enriched interrupt programs can be found in time linear in the size of the program and exponential in the number of handlers. The complexity of exact maximum stack size for monotonic enriched interrupt programs is PSPACE.*

In polynomial space one can generate in lexicographic order all the nodes which have a pc value of the starting statement of a handler. If the node is reachable from the start node and it is self-reachable then the stack size is reported as infinite. Since reachability is in NLOGSPACE and we have a exponential size of graph this can be achieved in NPSPACE. If in all the nodes the answer is "NO" then we need to find the longest path from the start node to the target. As longest path in a DAG is also in NLOGSPACE this can be achieved in NPSPACE.

It follows that both the stack boundedness and exact maximum stack size problems for monotonic enriched interrupt programs are PSPACE-complete.

5 Nonmonotonic Enriched Interrupt Programs

In this section we consider interrupt programs with tests, but do not restrict handlers to be monotonic. We give an EXPTIME algorithm to check stack boundedness and exact maximum stack size for this class of programs. The algorithm involves computing longest context free paths in context free DAGs, a technique that may be of independent interest.

5.1 Longest Paths in Acyclic Context Free Graphs

We define a context free graph as in [8]. Let Σ be a collection of k opening and closing parentheses, i.e., $\Sigma = \{(^1, (^2, \ldots, (^k,)^1,)^2, \ldots,)^k\}$. A *context free graph* is a tuple $G = (V, E, \Sigma)$ where

– V is a set of nodes.
– $E \subseteq (V \times V \times (\Sigma \cup \{\tau\}))$ is a set of labeled edges (and τ is a special symbol not in Σ).

We associate with each edge a *weight function* $wt : E \to \{0, +1, -1\}$ defined as follows:

– $wt(e) = 0$ if e is of the form (v, v', τ),
– $wt(e) = -1$ if e is of the form $(v, v',)^i)$ for some i,

– $wt(e) = 1$ if e is of the form $(v, v', (^i)$ for some i.

Let $\Sigma = \{(^1, (^2, \ldots, (^k,)^1,)^2, \ldots,)^k\}$ be an alphabet of matched parentheses. Let $\mathcal{L}$ be the language generated by the context free grammar

$$\begin{array}{l} M \rightarrow M(^i S \qquad \text{for } 1 \leq i \leq k \\ S \rightarrow \epsilon \quad | \quad (^i S)^i S \text{ for } 1 \leq i \leq k \end{array}$$

from the starting symbol M. Thus $\mathcal{L}$ defines words of matched parentheses with possibly some opening parentheses mismatched. A *context-free path* π in a context-free graph G is a sequence of vertices $v_1, v_2, \ldots, v_k$ such that for all $i = 1, \ldots, k-1$, there is an edge between v_i and v_{i+1}, i.e., there is a letter $\sigma \in \Sigma \cup \{\tau\}$ such that $(v_i, v_{i+1}, \sigma) \in E$ and the projection of the labels along the edges of the path on to Σ is a word in $\mathcal{L}$. Given a context free path π with edges $e_1, e_2, \ldots, e_k$ the *cost* of the path $Cost(\pi)$ is defined as $\sum_i wt(e_i)$. Note that $Cost(\pi) \geq 0$ for any context-free path π. A context-free graph G is a *context-free DAG* iff there is no cycle C of G such that $\sum_{e \in C} wt(e) > 0$. Given a context-free DAG $G = (V, E, \Sigma)$ we define an ordering $order : V \rightarrow \mathbb{N}$ of the vertices satisfying the following condition: if there is a path π in G from vertex v_i to v_j and $Cost(\pi) > 0$ then $order(v_j) < order(v_i)$. This ordering is well defined for context-free DAGs. Let G be a context free DAG G, and let $V = \{v_1, v_2, \ldots, v_n\}$ be the ordering of the vertex set consistent with $order$ (i.e., $order(v_i) = i$). We give an polynomial procedure to find the longest context-free path from any node v_i to v_1 in G.

Algorithm 1 Function LongestContextFreePath

Input: A context free DAG G, a vertex v_1 of G
Output: For each vertex v of G, return the length of the longest context free path from v to v_1, and 0 if there is no context free path from v to v_1

1. For all vertex $v_j \in V$
   $val[v_j] \leftarrow 0$
2. Construct the transitive closure matrix T such that
   $T[i, j] = 1$ iff there is a context free path from i to j
3. For $j \leftarrow 1$ to n
   3.1 For all immediate successor v_i of v_j such that the edge e_{v_j, v_i} to v_i satisfies $wt(e_{v_j, v_i}) \geq 0$
   $val[v_j] = \max\{val[v_j], val[v_i] + wt(e_{v_j, v_i})\}$
   3.2 For all $v_i \in V$
   3.2.1 if($T[i, j]$) (v_j is context-free reachable from v_i)
   $val[v_i] = \max\{val[v_i], val[v_j]\}$

The correctness proof of our algorithm uses a function Num from paths to $\mathbb{N}$. Given a path π we define $Num(\pi)$ as $\max\{order(v) \mid v \text{ occurs in } \pi\}$. Given a node v let $L_v = \{L_1, L_2, \ldots, L_k\}$ be the set of longest paths from v to v_1. Then we define $Num_{v_1}(v) = \min\{Num(L_i) : L_i \in L_v\}$. The correctness of the algorithm follows from the following set of observations. First, if there is a longest path L from a node v to v_1 such that L starts with an opening parenthesis $(^i$ that is not matched along the path L then $order(v) = Num_{v_1}(v)$. A node v in the DAG G satisfies the following conditions.

– If $Num_{v_1}(v) = order(v) = j$ then within the execution of Statement 3.1 of the j-th iteration of Loop 3 of Function LongestContextFreePath $val[v]$ is equal to the cost of a longest path from v to v_1.
– If $order(v) < Num(v) = j$ then by the j-th iteration of Loop 3 of Function LongestContextFreePath $val[v]$ is equal to the cost of a longest path from v to v_1.

Finally, notice that at the end of function LongestContextFreePath(G, v_1), for each vertex v, the value of $val[v]$ is equal to the longest context free path to v_1, and equal to zero if there is no context free path to v_1. In the Function LongestContextFreePath the statement 3.2.1 gets executed at most n^2 times since the loop 3 gets executed n times at most and the nested loop 3.2 also gets executed n times at most. The context-free transitive closure can be constructed in $O(|V|^3)$ time [11]. Hence the complexity of our algorithm is polynomial and it runs in time $O(n^2+n^3) = O(n^3)$. A detailed proof of correctness is given in Appendix C.

Theorem 5. *The longest context free path of a context free DAG can be found in time cubic in the size of the graph.*

To complete our description of the algorithm, we must check if a given context free graph is a context free DAG, and generate the topological ordering *order* for a context free DAG.

We give a polynomial time procedure to check given a context-free graph whether it is a DAG. Given a context-free graph $G = (V, E, \Sigma)$ let its vertex set be $V = \{1, 2, \ldots, n\}$. For every node $k \in V$ the graph G can be unrolled as a DAG for depth $|V|$ and it can be checked if there is a path π from k to k such that $Cost(\pi) > 0$. We give the construction of the DAG below.

Given the graph G and a node k we create a context-free DAG $G_k = (V_k, E_k, \Sigma)$ as follows:

1. $V_k = \{k_0\} \cup \{(i,j) : i \in \{1, 2, \ldots, n-2\}, j \in \{1, 2, \ldots, n\}\} \cup \{k_{n-1}\}$
2. $E_k = \{\langle k_0, (1,j), * \rangle : \langle k, j, * \rangle \in E\}$
   $\cup \{\langle (i,j), (i+1, j'), * \rangle : \langle j, j', * \rangle \in E\}$
   $\cup \{\langle (n-2, j), k_{n-1}, * \rangle : \langle j, k, * \rangle \in E\}$
   $\cup \{\langle k_0, (1,k) \rangle\} \ \cup \{\langle (i,k), (i+1,k) \rangle \ \}$

where $*$ can represent a opening parenthesis, closing parenthesis or can be τ.

Notice that the edges in the last line ensures if there is a cycle of positive cost from k to itself with length $t < n$ from then it is possible to go from k_0 to $(n-t-1, k)$ and then follow the path and reach k_{n-1} and hence the whole path has a positive cost.

In this graph we can find the longest-context free path from k_0 to k_n (by the Function LongestContextFreePath) and if the value is positive we know there is a positive cycle in G from k to k. If for all nodes the value of the longest path is 0 then G is context-free DAG and longest path can be evaluated in G.

Given a context-free DAG G we can find the $order(v)$ of the vertices in polynomial time. If a vertex v can reach v' and v' can reach v put them in the same group of vertices. Both the path from v to v' and v' to v must be cost 0 else there would be a cycle of positive cost. Hence the ordering of vertices within a group can be arbitrary. We can topologically order the graph induced by the groups and then assign an order to the vertices where vertices in the same group are ordered arbitrarily.

5.2 Stack Size Analysis

We present an algorithm to check for stack boundedness and exact maximum stack size. The idea is to perform context free longest path analysis on the state space of the program.

Given a program p of m statements and n handlers we label the statements as $pc_1, pc_2, \ldots, pc_m$. Let PC denote the set of all statements i.e $PC = \{pc_1, pc_2, \ldots, pc_m\}$. We construct a context-free graph [8] $G_p = \langle V, E, \Sigma \rangle$, called the *state graph of* p, where $\Sigma = \{(^1, (^2, \ldots, (^m,)^1,)^2, \ldots)^m\}$ as follows:

- $V = PC \times IMR$ where IMR denote all 2^n possible imr values.

– $E \subseteq (V \times V \times \Sigma) \cup (V \times V)$
  1. $(v, v', (_i) \in E$ iff $v = (pc_i, imr_1)$ and $v' = (pc_j, imr_2)$ and pc_j is the starting address of some handler which is enabled with imr imr_1 and $imr_2 = imr_1 \wedge \neg \mathtt{t}_0$. This models the call to a handler.
  2. $(v, v',)_i) \in E$ iff $v = (pc_i, imr_1)$ and $v' = (pc_j, imr_2)$ and pc_j is the return statement of some handler and $imr_1 = imr_2 \vee \mathtt{t}_0$. This models the return from a handler.
  3. $(v, v') \in E$ iff $v = (pc_i, imr_1)$ and $v' = (pc_j, imr_2)$ and executing the statement at pc_i with imr value imr_1 the control goes to pc_j and the imr value is imr_2. This models the execution of a statement.

The vertex $(pc_1, 000 \cdots 000)$ is the starting vertex of G_p. Let the reachable state graph G'_p be the induced sub-graph of G_p which contains only nodes which are context-free reachable from the start node. If G'_p is not a context-free DAG then we report that stack is unbounded. If G'_p is a context free DAG we create a new DAG G''_p adding a new vertex $target$ and adding edges to $target$ from all nodes of G'_p of weight 0 and find the value of the longest context-free path from the start vertex to $target$ in the DAG G''_p.

Algorithm 2 Function StackSizeGeneral

Input: Enriched interrupt driven program p
Output: $maxStackSize(P_p)$
1. Build the state graph G_p from the Program p
2. Let $V' = \{v' :$ there is a context free path from the starting vertex to $v'\}$
3. Let G'_p be the sub-graph of G_p induced by the vertex set V'.
4. If G'_p is not a context free DAG then return "infinite"
5. Else create $G''_p = (V'', E'', \Sigma)$ as follows :
   5.1 $V'' = V' \cup \{target\}$
   5.2 $E'' = E' \cup \{(v, target) : v \in V'\}$
6. Return the value of the longest context free path from the starting vertex to $target$

From the construction of the state graph, it follows that there is a context free path from a vertex $v = (pc, imr)$ to $v' = (pc', imr')$ in the state graph G_p iff there exists stores R, R' and stacks σ, σ' such that $\langle \bar{h}, R, imr, \sigma, pc \rangle \rightarrow^* \langle \bar{h}, R', imr', \sigma' pc' \rangle$. Moreover, if G'_p is the reachable state graph then there exists K such that for all P' such that $P_p \rightarrow^* P'$ we have $|P'.stk| \leq K$ iff G'_p is a context free DAG. To see this, first notice that if G'_p is not a context free DAG then there is a cycle of positive cost. Traversing this cycle infinitely many times makes the stack grow unbounded. On the other hand, if the stack is unbounded then there is a program address that is visited infinitely many times with the same imr value and the stack grows between the successive visits. Hence there is a cycle of positive cost in G'_p. These observations, together with Theorem 5 show that function StackSizeGeneral correctly computes the exact maximum stack size of an interrupt program p.

Theorem 6. *The exact maximum stack size of nonmonotonic enriched interrupt programs can be found in time cubic in the size of the program and exponential in the number of handlers.*

It is easy to notice the number of vertices in G_p is $m \times 2^n$ where we have m program statements and n interrupt handlers. It follows from Theorem 5 and discussion of the earlier section that all the step 1,2,3,4,5 and 6 of StackSizeGeneral can be computed in time polynomial in G_p. Since G_p is exponential in size of the input program p we have an EXPTIME procedure for Stack Boundedness and Exact Maximum Stack Size for nonmonotonic enriched interrupt programs. Notice also that our syntax ensures that all

statements that modify the imr are monotonic: if $imr_1 \leq imr_2$, and for $i = 1, 2$, we have $P(imr_i) \rightarrow P'(imr'_i)$ for any two program states $P(imr)$ and $P'(imr)$ parameterized by imr, then $imr'_1 \leq imr'_2$. In particular, we only allow bitwise or's on the imr if we test if a bit is set, and only allow bitwise and's on the imr if we test if a bit is unset. We can extend the syntax of the enriched calculus to allow any imr operations, but the above algorithm correctly solves the exact maximum stack size problem in this extended case, with no change in complexity.

Finally, we leave open whether the exact maximum stack size problem is EXPTIME-hard or PSPACE-complete (the problem is PSPACE-hard by Theorem 3).

6 Conclusion

We have presented NLOGSPACE, PSPACE, and EXPTIME algorithms for problems of importance to designers of real-time systems, together with lower bounds. It remains to be seen whether the PSPACE and EXPTIME algorithms can be useful in practice. One can note that the time to execute the algorithms seems to primarily grow with the *number* of interrupt handlers, and not so much with the *size* of the interrupt handlers. So, for example, for a Motorola Dragonball processor, which handles 22 interrupt sources, it may be feasible to run an algorithm that is exponential in the number of interrupt handlers. Experiments are needed to settle this.

Acknowledgments. Palsberg, Ma, and Zhao were supported by the National Science Foundation ITR award 0112628. Henzinger, Chatterjee, and Majumdar were supported by the AFOSR grant F49620-00-1-0327, the DARPA grants F33615-C-98-3614 and F33615-00-C-1693, the MARCO grant 98-DT-660, and the NSF grants CCR-0208875 and CCR-0085949.

References

1. Dennis Brylow, Niels Damgaard, and Jens Palsberg. Static checking of interrupt-driven software. In *Proceedings of ICSE'01, 23rd International Conference on Software Engineering*, pages 47–56, Toronto, May 2001.
2. Paul Hudak. *The Haskell School of Expression—Learning Functional Programming through Multimedia*. Cambridge University Press, 2000.
3. John Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types. In *Proceedings of POPL'96, 23nd Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 410–423, 1996.
4. Jens Palsberg and Di Ma. A typed interrupt calculus. In *The seventh International Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems*, Oldenburg, Germany, September 2002.
5. C.H. Papadimitriou. *Computational Complexity*. Addision-Wesley, 1994.
6. Lars Pareto. Sized types. M.S. Thesis, Chalmers University of Technology, 1996.
7. Lars Pareto. *Types for Crash Prevention*. PhD thesis, Chalmers University of Technology, 2000.
8. T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL 95: Principles of Programming Languages*, pages 49–61. ACM, 1995.
9. Zhanyong Wan, Walid Taha, and Paul Hudak. Real-time FRP. In *Proceedings of ICFP'01, ACM SIGPLAN International Conference on Functional Programming*, pages 146–156, 2001.
10. Zhanyong Wan, Walid Taha, and Paul Hudak. Even-driven FRP. In *Proceedings of PADL'02, Practical Aspects of Declarative Languages*, pages 155–172, 2002.
11. Mihalis Yannakakis. Graph theoretic methods in database theory. In *Proc. 9th ACM Symp. on Principles of Database Systems*, pages 203–242. 1990.

Appendix

A Exact Maximum Stack Size is PSPACE-hard for Monotonic Interrupt Programs

Lemma 3. *For a simple interrupt calculus program p, we have that $maxStackSize(P_p) = |\mathcal{M}(imr_S)|$.*

Proof. By definition, the state of a simple interrupt program $p = (m, \bar{h})$ is of the form $\langle \bar{h}, 0, imr, \sigma, a\rangle$ and stack size of p increases whenever an interrupt is handled and we have state transition of the form

$$\langle \bar{h}, 0, imr, \sigma, a\rangle \rightarrow \langle \bar{h}, 0, imr \wedge \neg t_0, a :: \sigma, \bar{h}(i)\rangle \text{ if } imr \geq t_i \vee t_0.$$

Let a_i, $i \in \{1..4\}$ represent the four statements in the body of an interrupt handler such that any handler is of the form $a_1; a_2; a_3; a_4$. By definition of simple interrupt program., the master bit is enabled only between a_2 and a_3, where calls to other handlers may occur. Also, after a call to a interrupt handler returns, the imr value is always less than or equal to the imr value before the call. Thus, during a call to handler h_i with initial imr value equal to imr, the only possible states where interrupts maybe be handled are of the form

$$\langle \bar{h}, 0, imr', \sigma, a_3; a_4\rangle, \text{where } imr' \leq f_i(imr).$$

Let $\rightarrow^*$ be the reflexive, transitive closure of $\rightarrow$. Then, we only need to examine state transitions of the following form to compute $maxStackSize(P_p)$.

$$\langle \bar{h}, 0, imr, \sigma, a\rangle \rightarrow^* \langle \bar{h}, 0, imr', a :: \sigma, a_3; a_4\rangle, \text{ where } imr' \leq f_i(imr)\ \forall i, \text{ such that } t_i \vee t_0 \leq imr.$$

Let $P = \langle \bar{h}, 0, imr, \sigma, a\rangle$ and $P' = \langle \bar{h}, 0, imr', \sigma, a\rangle$. We can show by induction that $maxStackSize(P) \leq maxStackSize(P')$ if $imr \leq imr'$.

Therefore, it is sufficient to consider state transitions of the form

$$\langle \bar{h}, 0, imr, \sigma, a\rangle \rightarrow^* \langle \bar{h}, 0, f_i(imr), a :: \sigma, a_3; a_4\rangle, \text{ where } imr \geq t_i \vee t_0.$$

In the main loop, the possible states where interrupts may be handled are of the form

$$\langle \bar{h}, 0, imr_S \vee t_0, \mathsf{nil}, \mathsf{loop\ skip}\rangle$$
$$\langle \bar{h}, 0, imr_S \vee t_0, \mathsf{nil}, \mathsf{skip; loop\ skip}\rangle.$$

Let a_0 be the statements of the form loop skip or skip;loop skip.

To compute $maxStackSize(P_p)$, we only need to consider transitions of the form

$\langle \bar{h}, 0, imr_S \vee t_0, \sigma, a_0\rangle \rightarrow^* \langle \bar{h}, 0, f_i(imr_S) \vee t_0, a_0 :: \sigma, a_3; a_4\rangle$, where $imr_S \geq t_i$, and
$\langle \bar{h}, 0, imr, \sigma, a_3; a_4\rangle \rightarrow^* \langle \bar{h}, 0, f_j(imr), a_3; a_4 :: \sigma, a_3'; a_4'\rangle$, where $imr \geq t_j \vee t_0$.

It is now clear that we can just use $imr \wedge \neg t_0$ to represent states that we are interested in with starting states represented by imr_S. The above two kinds of transitions can be uniquely represented by edges of the form $(imr_S, f_i(imr_S), i), (imr \wedge \neg t_0, f_j(imr \wedge \neg t_0), j)$ in graph $G(p)$.

Therefore, $maxStackSize(P_p)$ is equal to the length of the longest path in $G(p)$ from the start node imr_S. ∎

Lemma 4. *For a simple interrupt calculus program p, and a subgraph of $G(p)$, we have that if $imr \leq imr_1 \vee imr_2$, then $|\mathcal{M}(imr)| \leq |\mathcal{M}(imr_1)| + |\mathcal{M}(imr_2)|$.*

Proof. Claim 1: If $imr \leq imr_1 \vee imr_2$, and P is a path from node imr to imr', then we can find a path P_1 from imr_1 to imr_1' and a path P_2 from node imr_2 to imr_2' such that $|P| = |P_1| + |P_2|$ and $imr' \leq imr_1' \vee imr_2'$.

Proof of Claim 1. We proceed by induction on the length of P. The base case of $|P| = 0$ is trivially true. Suppose Claim 1 is true for $|P| = k$ and that P' is P appended with an edge to imr''. We need to prove the case of P'. Since P ends at imr', $\exists t_i \leq imr'$ such that $imr'' = f_i(imr')$. By induction hypothesis, $t_i \leq imr' \leq imr_1' \vee imr_2'$. Thus, $\exists a \in \{1,2\}$ such that $t_i \leq imr_a'$. Suppose that $t_i \leq imr_1'$ (the case of $t_i \leq imr_2'$ is similar and is omitted). We can let P_1' be P_1 appended with an edge to imr_1'' where $imr_1'' = f_i(imr_1')$. By definition of f_i, we have $f_i(imr') \leq f_i(imr_1') \vee imr_2'$. Thus, we have $|P'| = |P| + 1 = |P_1| + 1 + |P_2| = |P_1'| + |P_2|$ and $imr'' \leq imr_1'' \vee imr_2'$.

We can apply Claim 1 to the situation with $\mathcal{M}(imr)$ as the path P from imr to 0. Since $|\mathcal{M}(imr_1)| \geq |P_1|$ and $|\mathcal{M}(imr_2)| \geq |P_2|$, we have $|\mathcal{M}(imr)| \leq |\mathcal{M}(imr_1)| + |\mathcal{M}(imr_2)|$. ∎

Theorem 7. *The exact maximum stack size problem for monotonic interrupt programs is PSPACE-hard.*

Proof. We will do a reduction from the QSAT problem. Suppose we are given an instance of QSAT problem

$$S = \exists x_n \forall x_{n-1} \ldots \exists x_2 \forall x_1 \quad \phi,$$

where ϕ is a 3SAT instance in conjunctive normal form of n variables $x_n, \ldots, x_1$ and L boolean clauses. Let ϕ_{ij} be the jth literal of the ith clause in ϕ and $\phi = \bigwedge_{i=1}^{L} \bigvee_{j=1}^{3} \phi_{ij}$.

We construct a program $p = (m, \bar{h})$ and $\bar{h} = \{h(i) \mid i \in \{1 \ldots 3L + 4n\}\}$.

As before, we define a graph $G(p) = (V, E)$ such that $V = \{imr \mid imr(0) = 0\}$ and $E = \{(imr, f_i(imr), i) \mid t_i \leq imr\}$, where $f_i(imr) = imr \wedge \neg imr' \vee imr''$ iff $h(i) = \mathcal{H}(imr';\ imr'')$.

For clarity, we define three kinds of indices: $d_{ij} = 3(i-1)+j$, where $i \in \{1..L\}, j \in \{1..3\}$; $q_i^a = 3L + 4i - 3 + a$, and $w_i^a = 3L + 4i - 1 + a$, where $i \in \{1..n\}, a \in \{0,1\}$.

Let

$$\begin{aligned}
D &= \{d_{i1}, d_{i2}, d_{i3} \mid \forall i \in \{1..L\}\} \\
D_{ij} &= \{d_{i1}, d_{i2}, d_{i3}\} \cup \{w_k^a \mid (a = 1 \wedge \phi_{ij} = x_k) \vee (a = 0 \wedge \phi_{ij} = \neg x_k)\} \\
W_i &= \{w_j^a \mid \forall j \in \{1..i\}, \forall a \in \{0,1\}\} \\
Q_i &= \{q_j^a \mid \forall j \in \{1..i\}, \forall a \in \{0,1\}\}.
\end{aligned}$$

We will use the abbreviation

$$imr_0 = \bigvee_{i \in D} t_i, \quad imr_k = t_{q_k^0} \vee t_{q_k^1} \ \forall k \in \{1..n\}.$$

Assume that n is even. For all $a \in \{0,1\}$, let

$$\begin{aligned}
f_{q_{2k-1}^a}(x) &= x \wedge \neg t_{q_{2k-1}^a} \vee (imr_{2k-2} \vee t_{w_{2k-1}^a}), && \forall k \in \{1..n/2\} \\
f_{q_{2k}^a}(x) &= x \wedge \neg imr_{2k} \vee (imr_{2k-1} \vee t_{w_{2k}^a}), && \forall k \in \{1..n/2\} \\
f_{w_k^a}(x) &= x \wedge \neg \bigvee_{i \in D \cup Q_{k-1} \cup W_k} t_i, && \forall k \in \{1..n\} \\
f_{d_{ij}}(x) &= x \wedge \neg \bigvee_{k \in D_{ij}} t_k, && \forall i \in \{1..L\}, j \in \{1,2,3\}.
\end{aligned}$$

Given an imr value r, we define the graph $G_r(p)$ to be the subgraph of $G(p)$ such that any edge labeled d_{ij} is removed for all i, j such that $\phi_{ij} = \neg x_k$ and $t_{w_k^0} \leq r$, or $\phi_{ij} = x_k$ and $t_{w_k^1} \leq r$. We use $\mathcal{M}_r(imr)$ to denote the longest path in $G_r(p)$ from imr.

Claim 1: $\forall k \in \{1..\frac{n}{2}\}$, $|\mathcal{M}_r(imr_{2k})| = \max_{a\in\{0,1\}} |\mathcal{M}_r(t_{q^a_{2k}})|$, and $|\mathcal{M}_r(imr_0)| \leq L$.

Proof of Claim 1. By definition, we have that $\forall a \in \{0,1\}$, $f_{q^a_{2k}}(x) = x \wedge \neg imr_{2k} \vee (imr_{2k-1} \vee t_{w^a_{2k}})$, from which the claim follows.

By definition of $f_{d_{ij}}$, for each $i \in \{1..L\}$, $\mathcal{M}(imr_0)$ can contain at most one edge with label d_{ij}, where $j \in \{1,2,3\}$. Thus, $|\mathcal{M}(imr_0)| \leq L$.

Claim 2: $|\mathcal{M}_r(imr_{2k-1})| = \sum_{b\in\{0,1\}} |\mathcal{M}_r(t_{q^b_{2k-1}})|$.

Proof of Claim 2. From Lemma 4, we have $|\mathcal{M}_r(imr_{2k-1})| \leq \sum_{b\in\{0,1\}} |\mathcal{M}_r(t_{q^b_{2k-1}})|$.

Let P be the path from imr_{2k-1} to $t_{q^1_{2k-1}}$ constructed from $\mathcal{M}_r(t_{q^0_{2k-1}})$ by replacing any node imr on $\mathcal{M}_r(t_{q^0_{2k-1}})$ with $imr \vee t_{q^1_{2k-1}}$. It is straightforward to show that if edge (imr, imr', i) is on $\mathcal{M}_r(t_{q^0_{2k-1}})$, then $(imr \vee t_{q^1_{2k-1}}, imr' \vee t_{q^1_{2k-1}}, i)$ is on P. If we concatenate P with $\mathcal{M}_r(t_{q^1_{2k-1}})$, then we have a path from imr_{2k-1} of length $|\mathcal{M}_r(t_{q^0_{2k-1}})|+|\mathcal{M}_r(t_{q^1_{2k-1}})|$.

Claim 3: $|\mathcal{M}(imr_n)| \leq 2^{n/2}(6+L) - 6$.

Proof of Claim 3. It is sufficient to prove that $|\mathcal{M}(imr_{2k})| \leq 2^k(6+L) - 6$. For all $a \in \{0,1\}$ we have:

$$\begin{aligned}
|\mathcal{M}(t_{q^a_{2k}})| &= 1 + |\mathcal{M}(imr_{2k-1} \vee t_{w^a_{2k}})| \\
&\leq 2 + |\mathcal{M}(imr_{2k-1})| \\
&= 2 + \sum_{b\in\{0,1\}} |\mathcal{M}(t_{q^b_{2k-1}})| \\
&= 4 + \sum_{b\in\{0,1\}} |\mathcal{M}(imr_{2k-2} \vee t_{w^b_{2k-1}})| \\
&\leq 6 + 2|\mathcal{M}(imr_{2k-2})|
\end{aligned}$$

$$\begin{aligned}
|\mathcal{M}(imr_{2k})| &= \max_{a\in\{0,1\}} (|\mathcal{M}(t_{q^0_{2k}})|, |\mathcal{M}(t_{q^1_{2k}})|) \\
&\leq 6 + 2|\mathcal{M}(imr_{2k-2})|
\end{aligned}$$

From the last inequality and Claim 1, it is straightforward to show the claim by induction on k.

Claim 4: For any r and $a \in \{0,1\}$, $|\mathcal{M}_r(t_{q^a_{2k}})| = 2^k(6+L) - 6$ iff $\forall b \in \{0,1\}$, $|\mathcal{M}_{r'}(t_{q^b_{2k-1}})| = 2^{k-1}(6+L) - 4$, where $r' = r \vee t_{w^a_{2k}}$.

Proof of Claim 4. Suppose that $|\mathcal{M}_r(t_{q^a_{2k}})| = 2^k(6+L) - 6$. The path $\mathcal{M}_r(t_{q^a_{2k}})$ must contain the edge with label w^a_{2k} because otherwise

$$\begin{aligned}
|\mathcal{M}_r(t_{q^a_{2k}})| &= 1 + |\mathcal{M}_r(imr_{2k-1} \vee t_{w^a_{2k}})| \\
&= 1 + |\mathcal{M}_r(imr_{2k-1})| \\
&\leq 1 + |\mathcal{M}(imr_{2k-1})| \leq 2^k(6+L) - 7.
\end{aligned}$$

By definition of $f_{q^a_{2k}}$, for any node imr on the path $\mathcal{M}_r(imr_{2k-1} \vee t_{q^a_{2k}})$, we have $f_{w^a_{2k}}(imr) = 0$. Thus, the edge labeled w^a_{2k} can only be the last edge on $\mathcal{M}_r(t_{q^a_{2k}})$. By definition of $f_{d_{ij}}$, the longest path from $imr_{2k-1} \vee t_{w^a_{2k}}$ containing edge labeled w^a_{2k} does not contain any edge labeled d_{ij} for all i, j such that $\phi_{ij} = x_{2k}$ if $a = 1$, and $\phi_{ij} = \neg x_{2k}$ if $a = 0$. This path

is the same path in $G_{r'}(p)$, where $r' = r \vee t_{w^a_{2k}}$. Therefore,

$$\begin{aligned}
|\mathcal{M}_r(t_{q^a_{2k}})| = 2^k(6+L) - 6 &= |\mathcal{M}_{r'}(t_{q^a_{2k}})| \\
&= 1 + |\mathcal{M}_{r'}(imr_{2k-1} \vee t_{w^a_{2k}})| \\
&\leq 1 + |\mathcal{M}_{r'}(imr_{2k-1})| + |\mathcal{M}_{r'}(t_{w^a_{2k}})| \\
&= 2 + \sum_{b\in\{0,1\}} |\mathcal{M}_{r'}(t_{q^b_{2k-1}})|, \text{ and} \\
\sum_{b\in\{0,1\}} |\mathcal{M}_{r'}(t_{q^b_{2k-1}})| &\geq 2^k(6+L) - 8.
\end{aligned}$$

Since

$$\begin{aligned}
|\mathcal{M}_{r'}(t_{q^b_{2k-1}})| &= 1 + |\mathcal{M}_{r'}(imr_{2k-2} \vee t_{w^b_{2k-1}})| \\
&\leq 2 + |\mathcal{M}_{r'}(imr_{2k-2})| \\
&\leq 2 + |\mathcal{M}(imr_{2k-2})| = 2^{k-1}(6+L) - 4,
\end{aligned}$$

we have $\forall b \in \{0,1\}$, $|\mathcal{M}_{r'}(t_{q^b_{2k-1}})| = 2^{k-1}(6+L) - 4$.

Conversely, assume that for all $b \in \{0,1\}$, $|\mathcal{M}_{r'}(t_{q^b_{2k-1}})| = 2^{k-1}(6+L) - 4$ where $r' = r \vee t_{w^a_{2k}}$. From Claim 2, we know that $|\mathcal{M}_{r'}(imr_{2k-1})| = \sum_{b\in\{0,1\}} |\mathcal{M}_{r'}(t_{q^b_{2k-1}})| = 2^k(6+L) - 8$.

Let P be a path from $imr_{2k-1} \vee t_{w^a_{2k}}$ to $t_{w^a_{2k}}$ constructed from $\mathcal{M}_{r'}(imr_{2k-1})$ by replacing any node imr on $\mathcal{M}_{r'}(imr_{2k-1})$ with $imr \vee t_{w^a_{2k}}$. It is straightforward to show that if edge (imr, imr', i) is on $\mathcal{M}_{r'}(imr_{2k-1})$, then $(imr \vee t_{w^a_{2k}}, imr' \vee t_{w^a_{2k}}, i)$ is on P as well.

If we concatenate P with $\mathcal{M}_{r'}(t_{w^a_{2k}})$, then we have a path from $imr_{2k-1} \vee t_{w^a_{2k}}$ in graph $G_{r'}(p)$ of length $2^k(6+L) - 7$. Thus, $|\mathcal{M}_r(t_{q^a_{2k}})| = |\mathcal{M}_{r'}(t_{q^a_{2k}})| = 2^k(6+L) - 6$.

Claim 5: for any r' and $b \in \{0,1\}$, $|\mathcal{M}_{r'}(t_{q^b_{2k-1}})| = 2^{k-1}(6+L)-4$ iff $|\mathcal{M}_{r''}(imr_{2k-2})| = 2^{k-1}(6+L) - 6$. where $r'' = r' \vee t_{w^b_{2k-1}}$,

Proof of Claim 5. The proof is similar to the proof of Claim 4, we omit the details.

Claim 6: $|\mathcal{M}(imr_n)| = 2^{n/2}(6+L) - 6$ iff $\exists a_n \forall a_{n-1} \ldots \exists a_2 \forall a_1 \in \{0,1\}$, such that for $r = \bigvee_{k\in\{1..n\}} t_{w^{a_k}_k}$, $|\mathcal{M}_r(imr_0)| = L$.

Proof of Claim 6. From Claims 1+3, we know that $|\mathcal{M}_r(imr_{2k})| = 2^k(6+L) - 6$ iff $\exists a \in \{0,1\}$ such that $|\mathcal{M}_r(t_{q^a_{2k}})| = 2^k(6+L) - 6$. Together with Claim 4 and Claim 5, we have that for $k \in \{1..n/2\}$,

$$\begin{aligned}
&|\mathcal{M}_r(imr_{2k})| = 2^k(6+L) - 6, \text{ iff } \exists a \in \{0,1\}, \forall b \in \{0,1\}, \text{ such that} \\
&|\mathcal{M}_{r''}(imr_{2k-2})| = 2^{k-1}(6+L) - 6, \text{ where } r'' = r \vee t_{w^a_{2k}} \vee t_{w^b_{2k-1}}.
\end{aligned}$$

It is straightforward to prove by induction from $k = n/2$ to 1, that

$$\begin{aligned}
&|\mathcal{M}(imr_n)| = 2^{n/2}(6+L) - 6 \text{ iff } \exists a_n \forall a_{n-1} \ldots \exists a_{2k} \forall a_{2k-1}, \\
&\text{such that } |\mathcal{M}_r(imr_{2k-2})| = 2^{k-1}(6+L) - 6, \text{ where } r = \textstyle\bigvee_{i=2k-1}^{n} t_{w_i^{a_i}}.
\end{aligned}$$

The claim follows when $k = 1$.

Claim 7: S is satisfiable iff $\exists a_n \forall a_{n-1} \ldots \exists a_2 \forall a_1 \in \{0,1\}$, we have $|\mathcal{M}_r(imr_0)| = L$, where $r = \bigvee_{k=1}^{n} t_{w^{a_k}_k}$.

Proof of Claim 7. It is sufficient to prove that ϕ is satisfiable iff $\exists a_n, \ldots, a_1 \in \{0,1\}$, such that for $r = \bigvee_{k=1}^{n} t_{w^{a_k}_k}$, we have $|\mathcal{M}_r(imr_0)| = L$.

Suppose we have $a_n, \ldots, a_1$ such that $r = \bigvee_{k=1}^{n} t_{w_k^{a_k}}$, $|\mathcal{M}_r(imr_0)| = L$. We can construct a truth assignment T by defining $T(x_k) = true$ if $a_k = 0$ and $T(x_k) = false$ if $a_k = 1$. By definition of $f_{d_{ij}}$, for each $i \in \{1..L\}$, there exists a j such that the edge labeled d_{ij} is on $\mathcal{M}_r(imr_0)$. By definition of $\mathcal{M}_r$, if an edge labeled d_{ij} is on $\mathcal{M}_r(imr_0)$ and $\phi_{ij} = x_k$, then $a_k = 0$, and $T(x_k) = true$; and if $\phi_{ij} = \neg x_k$, then $a_k = 1$ and $T(x_k) = false$. $T(\phi_{ij}) = true$ in both cases. Therefore, T satisfies ϕ.

Conversely, suppose T satisfies ϕ. We can construct $r = \bigvee_{k=1}^{n} t_{w_k^{a_k}}$ from T by defining $a_k = 0$ if $T(x_k) = true$ and $a_k = 1$ if $T(x_k) = false$. For each $i \in \{1..L\}$, there exists j such that $T(\phi_{ij}) = true$, which means that the edge labeled d_{ij} can be on the path $\mathcal{M}_r(imr_0)$. Therefore, $|\mathcal{M}_r(imr_0)| = L$.

We conclude

S is not satisfiable
iff we do not have $\exists a_n \forall a_{n-1} \ldots \exists a_2 \forall a_1$,
such that for $r = \bigvee_{k=1}^{n} t_{w_k^{a_k}}$, $|\mathcal{M}_r(imr_0)| = L$ (Claim 7)
iff $|\mathcal{M}(imr_S)| < 2^{n/2}(6 + L) - 7$, where $imr_S = imr_n$ (Claims 3+6)
iff $maxStackSize(P_p) < 2^{n/2}(6 + L) - 7$ (Lemma 3),

so the exact maximum stack size problem is co-PSPACE-hard, that is, PSPACE-hard. ∎

B Stack Boundedness is PSPACE-hard for Monotonic Enriched Interrupt Programs

Fix a PSPACE-complete Turing Machine M which on any input x uses at most $r(|x|)$ space to decide whether M accepts x, where r is a polynomial. Given any input x, the TM M always halts and answers *accept* or *reject*. It is PSPACE-complete to decide given M and x whether $M(x)$ accepts or rejects [5]. Let the states of M be $Q = \{q_1, q_2, \ldots, q_t\}$ with q_t as the accepting state. Given such a machine M and an input x we reduce the problem of whether q_t is reachable to the stack boundedness analysis of a interrupt program such that the stack size is infinite iff q_t is reachable. We construct, from M and x a monotone interrupt program $p(M, x)$ such that M accepts x iff the stack of $p(M, x)$ is unbounded.

Now we describe the imr and handlers.

- The imr has one bit for master bit (the 0th bit).
- The imr has two bits for every position of the tape
- The imr has one bit for every (i, q, σ) which is turned on when the head of M is in the i-th cell, the control is in state q and σ is written in the ith cell. (σ is 0 or 1)

Total no of bits in the imr is $1 + 2 \times r(|x|) + 2 \times |Q| \times r(|x|)$. The first bit or the 0th is the master bit, the next $2 \times r(|x|)$ bits encodes what is in the tape cell of the TM M.

- $\forall k,\ 1 \leq\ k\ \leq r(|x|)$, $2 \times k - 1$-th bit is 1 and $2 \times k$-th bit is 0 if the tape cell in the k-th position of M is 0.
- $\forall k,\ 1 \leq\ k\ \leq r(|x|)$, $2 \times k - 1$-th bit is 0 and $2 \times k$-th bit is 1 if the tape cell in the k-th position of M is 1.
- $2 \times r(|x|) + 2 \times k \times r(|x|) + 2 \times i - 1$-th bit is 1 if the head of TM M is in position i, it is 0 there and the control is in state q_k. In the code for this handler we write the transition for the TM M corresponding $(q_k, 0)$.
- $2 \times r(|x|) + 2 \times k \times r(|x|) + 2 \times i$-th bit is 1 if the head of TM M is in position i, it is 1 there and the control is in state q_k. In the code for this handler we write the transition for the TM M corresponding $(q_k, 1)$.

The first $2 \times r(|x|)$ handlers which encode the tape cells do not change the status of the imr.

```
Handler Hi
 {
   iret
 }
```

We show how to write the TM M transition in the code for the handler. In the code below we use some short-hand notation which we explain below:

- write(σ, i) :: Write σ in the i-th cell tape of M which is achieved as follows:
  1. $\sigma = 0 ::= imr = imr \ \wedge \ \neg t_{2\times i}; imr = imr \ \vee \ t_{2\times i-1}$
     (Recall that t_j denotes the imr with all bits 0's and only the j-th bit 1.)
  2. $\sigma = 1 ::= imr = imr \ \wedge \ \neg t_{2\times i-1}; imr = imr \ \vee \ t_{2\times i}$.
- set(i-th bit on) :: Set the i-th bit of the imr on which is achieved by : $imr = imr \vee t_i$

- set(i-th bit off) :: Set the i-th bit of the imr off which is achieved by : $imr = imr \wedge \neg t_i$

1. Consider Handler H_j where $j = 2 \times r(|x|) + 2 \times k \times r(|x|) + 2 \times i - 1$ and the transition for $(q_k, 0)$ is $(q_{k'}, \sigma, R)$. Let us denote $l = 2 \times r(|x|) + 2 \times k' \times r(|x|) + 2 \times (i+1)$

   Code B1 Handler H_j

   {
   1.set(j-th bit off);
   2. write (σ, i)
   3.if(test $(2 * (i+1))$*-th bit 1)*
   *(*Check if the $i+1$-th TM cell is 1 *)*
   {
   3.1 set (l-th bit on)
   3.2 set (master bit on)
   }

   4.if(test $(2 * (i+1) - 1)$*-th bit 1)*
   *(*Check if the $i+1$-th TM cell is 0 *)*
   {
   4.1 set $(l-1$*-th bit on)*
   4.2 set (master 1 *bit on)*
   }

   5. $imr = imr \wedge 000\cdots000$
   6. iret
   }

   Note that the two consecutive imr_{or} statements in line 3.1 and 3.2 and 4.1 and 4.2 can be folded into a single imr_{or} statement.
   Similarly we can write codes for other transition types as well.
2. We now show how to write code for a handler H_j which corresponds to the accepting state

   Code B2 Handler H_j

{
1. set master bit on
2. iret
}

3. Code for the main program is as follows:

Code B3 Main

{
1. imr= imr $\vee$ *c*
where c is an imr constant which correctly encodes the starting configuration of M *on* x.
2. loop
skip
}

Lemma 5. *If* M *accepts* x *then the stack of* $p(M,x)$ *grows unbounded.*

Proof. We show that there is a sequence of occurrence of interrupts such that a handler corresponding to the accepting state is called. Whenever the l-th or the $l-1$-th bit in turned on and the master bit is turned on in line 3.1,3.2 or 4.1,4.2 an interrupt of type l or $l-1$ occurs. Hence following this sequence a handler corresponding to the accepting state is called and then the stack can grow unbounded as the handler sets the master bit on without disabling itself. ∎

Lemma 6. *If* M *does not accept* x *then the stack of* $p(M,x)$ *is bounded.*

Proof. If any handler which encodes the transitions of the $M(x)$ returns it sets all the bits of *imr* to 0 (Statement 6). Hence all the following checks in the statements 3 and 4 will fail and the master bit will not be set any further. Hence the stack would go empty. So if the stack is unbounded then no handler which encodes the configuration of the machine M returns. This proves that if the accepting state is not reached then the stack is bounded. ∎

From Lemmas 5, 6 we get the following result.

Theorem 8. *The stack boundedness problem for monotonic enriched interrupt programs is PSPACE-hard.*

C Correctness of Algorithm LongestContextFreePath

Lemma 7. *If there is a longest path* L *from a node* v *to* v_1 *such that* L *starts with an opening parenthesis* $(^i$ *that is not matched along the path* L *then* $order(v) = Num_{v_1}(v)$.

Proof. Consider any node v' in the path L. Since the first opening parenthesis is never matched, the sub-path $L(v,v')$ of L from v to v' satisfies $Cost(L(v,v')) > 0$. Hence it follows that for all nodes v' in L, we have $order(v') < order(v)$. Thus $Num_{v_1}(v) = order(v)$. ∎

Lemma 8. *A node* v *in the DAG* G *satisfies the following conditions.*

- *If* $Num_{v_1}(v) = order(v) = j$ *then within the execution of Statement 3.1 of the* j*-th iteration of Loop 3 of Function LongestContextFreePath* $val[v]$ *is equal to the cost of a longest path from* v *to* v_1.
- *If* $order(v) < Num(v) = j$ *then by the* j*-th iteration of Loop 3 of Function LongestContextFreePath* $val[v]$ *is equal to the cost of a longest path from* v *to* v_1.

Proof. We prove by induction on $Num(v)$. The base case holds when $Num(v) = 1$, since $v = v_1$. We now prove the inductive case. We consider the two cases when $order(v) = Num_{v_1}(v)$ and when $order(v) < Num_{v_1}(v)$.

Case $order(v) = Num_{v_1}(v) = j$. If the value of the longest path is 0 then it was fixed initially and it cannot decrease. Else there is a positive cost longest path from v to v_1. Let $L(v, v_1)$ be the longest path from v to v_1 such that $Num(L(v, v_1)) = order(v)$. We consider the two possible cases.

1. The longest path $L(v, v_1)$ is such that it starts with a opening parenthesis which is never matched. Let v'' be the successor of v in $L(v, v_1)$. Hence $order(v'') < order(v) = Num(v) = j$. Also the sub-path of $L(v'', v_1)$ of $L(v, v_1)$ is a longest path from v'' to v_1 (since otherwise we could have a greater cost path from v to v_1 by following the path from v to v'' and then the path from v'' to v_1). Hence $Num_{v_1}(v'') < Num_{v_1}(v) = j$. By the induction hypothesis, before the j-th iteration $val[v'']$ is equal to the longest path from v'' to v_1. Hence during the j-th iteration of Loop 3, when the loop of statement 3.1 is executed and v'' is chosen as v's successor then $val[v]$ is set to the cost of the longest path from v to v_1.
2. The longest path $L(v, v_1)$ goes through a node v' such that the cost of the subpath of $L(v, v')$ of $L(v, v_1)$ satisfies $Cost(L(v, v')) = 0$ and there is a opening parenthesis from v' which is not matched in $L(v, v_1)$. Clearly the sub-path $L(v', v_1)$ must be a longest path from v' to v_1 as otherwise $L(v, v_1)$ would not have been a longest path. It follows from Lemma 7 that $Num_{v_1}(v') = order(v) = k < j$. By hypothesis by the end of Statement 3.1 of k-th iteration $val[v']$ is equal to the longest path from v' to v_1. As v can context-free reach v' we have during the execution of statement 3.2 of the k-th iteration $val[v]$ is equal to the longest path from v to v_1.

Case $order(v) < Num_{v_1}(v) = j$ The longest path $L(v, v_1)$ goes through a node v' such that the cost of the subpath of $L(v, v')$ of $L(v, v_1)$ satisfies $Cost(L(v, v')) = 0$ and there is an opening parenthesis from v' which is not matched in $L(v, v_1)$. Clearly the sub-path $L(v', v_1)$ must be a longest path from v' to v_1 as otherwise $L(v, v_1)$ would not have been a longest path. It follows from Lemma 7 that $Num(v') = order(v) = k$. By hypothesis by the end of Statement 3.1 of k-th iteration $val[v']$ is equal to the longest path from v' to v_1. As v can context-free reach v' we have during the execution of statement 3.2 of the k-th iteration $val[v]$ is equal to the longest path from v to v_1. As $Num_{v_1}(v) \geq order(v')$ (since v' occurs in the path) it follows by the end of j-th iteration $val[v]$ is equal to the longest path from v to v_1.

∎

The Function LongestContextFreePath correctly evaluates the value of the longest path from a node v to v_1 for all nodes in the context-free DAG.

Lemma 9. *At the end of function* LongestContextFreePath(G, v_1)*, for each vertex v, the value of $val[v]$ is equal to the longest context free path to v_1, and equal to zero if there is no context free path to v_1. The function LongestContextFreePath can be evaluated in time $O(n^2 + T(n))$ where $T(n)$ is the time complexity of computing transitive closure of a context-free DAG.*

Proof. Correctness follows from Lemmas 7, 8. In the Function LongestContextFreePath the statement 3.2.1 gets executed at most n^2 times since the loop 3 gets executed n times at most and the nested loop 3.2 also gets executed n times at most. Since context-free

reachability is polynomial time [8] $T(n)$ is polynomial in n. The context-free transitive closure can be constructed in $O(|V|^3)$ time [11]. Hence the complexity of our algorithm is polynomial and it runs in time $O(n^2 + n^3) = O(n^3)$. ∎