20-2-R04

C 프로그램에서 사용되지 않는 자료를 찾는 정적 분석기의 개발 [1]

Development of Static Analyzer Detecting Unused Data in C Programs

황의권 · 이광근
서울대학교 컴퓨터공학부 프로그래밍 연구실
{neo; kwang}@ropas.snu.ac.kr

요 약

우리는 Airac5의 틀을 기반으로 C 프로그램에서 사용되지 않는 자료를 찾는 정적 프로그램 분석기 Umirac을 고안하고 구현하였다. Umirac은 구간 집합 도메인을 이용한 요약해석을 통하여 프로그램에서 정의되었지만 사용되지 않는 변수, 구조체의 필드, 그리고 버퍼의 구간에 대한 정보를 분석한다. 우리는 리눅스 커널 프로그램과 임베디드 시스템 소프트웨어를 대상으로 실험을 수행하여 실제 프로그램 상에서 불필요하게 정의된 자료들을 찾을 수 있었다. 또한 Umirac과 Airac7의 수행속도를 비교한 결과 구간 집합 분석을 하는 Umirac이 구간 분석을 하는 Airac7의 60–86%정도의 속도를 보여주었다.

1 서론

프로그램에서 메모리에 존재하지만 사용되지 않는 자료들은 프로그램의 수행성능을 저하시킨다. 이는 특히 임베디드 시스템과 같이 자원의 제약이 심한 환경에서 메모리 부족으로 인한 비정상적인 프로그램 종료를 일으키기도 하며, 불필요한 계산을 증가시켜 수행 속도에 영향을 끼치기도 한다.

프로그램의 수행성능을 향상시키기 위해서는 사용되지 않는 자료와 그것에 관련된 연산을 실행 전에 안전하게 제거하는 것이 필요하다. 컴파일러에서 코드 최적화 기법으로 널리 사용되는 살아있는 변수 분석(live variable analysis)[1]과 같은 방법은 컴파일 시간에 사용되지 않는 자료를 없애거나 레지스터를 효율적으로 사용하는 데에 유용하게 쓰이고 있다. 그러나 살아있는 변수 분석 자체는 포인터에 대한 분석을 포함하고 있지 않아 프로그램 실행 중에 각 포인터가 가리키는 대상을 알기 위해서는 별도의 포인터 분석을 사용하여야 하며, 버퍼에서 사용되지 않는 부분을 고려하지 못한다는 문제가 있다. SPLINT[9]는 버퍼 오버런과 같은 프로그램의 오류나 쓰이지 않는 변수, 구조체의 필드와 같은 불필요한 자료들을 찾아주는 분석기이다. 그러나 SPLINT는 변수나 구조체 필드에 값이 정의만 된 경우, 단순히 변수나 구조체 필드의 주소값을 취한 경우에도 해당 메모리 공간을 사용한 것으로 간주한다.

[1]본 연구는 정보통신부 선도기반기술개발사업과 교육인적자원부 두뇌한국21사업의 지원으로 수행하였다.

Umirac[우미락]은 요약해석 틀(abstract interpretation framework)[3, 4, 2]에 기반하여 C 프로그램[5]에서 사용되지 않는 자료를 찾아주는 정적 분석기이다. Umirac은 세 가지의 정보를 사용자에게 알려준다:

– 정의되었으나 사용되지 않는 변수(UV : unused variable)
– 사용되지 않는 구조체의 필드(UF : unused field)
– 사용되지 않는 버퍼 구간(BB : bubble in buffer)

Umirac의 설계와 구현은 Airac5[8]의 틀을 바탕으로 이루어졌다. 즉, Umirac은 Airac5와 마찬가지로 프로그램이 실행 중에 겪을 수 있는 모든 상황을 포섭하며, 정확도와 속도를 향상시키기 위한 많은 기술들이 적용되어 있다. Airac5의 구간 도메인을 분석에 보다 적합한 구간 집합 도메인으로 확장하였고, 요약 메모리에서 특정 메모리 공간이 사용되었는지의 여부를 표현하는 요소를 추가하였다.

2 분석기 설계

Umirac은 Airac5에서 사용한 G[7]의 요약해석을 구간 집합 도메인(set of interval domain)으로 확장한 설계를 사용하였다. 구간 분석(interval analysis)[3]이 사용되지 않는 버퍼 구간을 찾는 분석에서 유용하지 않은 예는 다음과 같다:

```
if (rand()%2)
    index = 0;
else index = ARRAY_SIZE - 1  /* 9 */;
return array[index];
```

구간 분석을 사용하였을 경우 마지막 `return`문에서의 `index`의 값은 [0, 9]로 요약된다. 만약 프로그램의 다른 부분에서 버퍼 `array`를 사용하는 부분이 없다면 버퍼 `array`에 사용하지 않는 구간 [1, 8]이 있음은 명백하지만 구간 분석으로는 이러한 정보를 알아낼 수 없다.

구간 집합 도메인은 요약 세계에서의 값을 정수 구간의 집합으로 표현한다. 예를 들어, 위의 예제 마지막 줄에서의 `index`의 값은 구간 집합 {[0, 0], [9, 9]}로 요약된다. 따라서 `array`에는 사용하지 않는 구간 [1, 8]이 있음을 알아낼 수 있다.

구간들의 집합을 $\hat{\mathbb{Z}}$라고 할 때, 구간 집합 도메인 $\hat{Int}$는 다음과 같이 정의할 수 있다:

$$2^{\mathbb{Z}} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} 2^{\hat{\mathbb{Z}}} = \hat{Int}$$

$$\begin{aligned} \gamma\, A &= \bigcup\{\gamma_{\hat{\mathbb{Z}}}\, a \mid a \in A\} \\ \hat{\bot}_{\hat{Int}} &= \emptyset \\ \hat{\top}_{\hat{Int}} &= \{[-\infty, \infty]\} \end{aligned}$$

여기서 두 구간 집합 간의 순서(order)는 다음과 같이 정의된다:

$$A, B \in \hat{Int}, A \sqsubseteq B \iff (\forall a \in A. \exists b \in B \,:\, a \sqsubseteq b)$$

이러한 구간 집합의 정의에 있어서 중요한 것은, 한 구간 집합의 원소인 구간들 사이에는 겹치거나 이어질 수 있는 관계가 존재하면 안 된다는 것이다. 이러한 성질이 유지되도록 각 연산의 마지막 단계에서 구간 집합을 정규화(normalize)할 필요가 있다. 예를 들어 두 구간 집합 A와 B의 합치기(join) 연산은 다음과 같이 정의할 수 있다:

$$A \sqcup B = |(A \cup B)|$$

정규화는 구간 집합 안에서 서로 겹치거나 이어질 수 있는 구간들을 하나의 구간으로 합치는 역할을 한다. 두 구간 집합 A와 B의 일반적인 연산은 다음과 같이 두 구간 집합이 포함하는 각 구간들의 연산 결과를 모으는 형태로 정의된다.

$$\hat{\diamond} \in \{\hat{+}, \hat{-}, \hat{*}, \hat{/}, \sqcap\} \ : \ \hat{Int} \times \hat{Int} \to \hat{Int}$$
$$\hat{\underline{\diamond}} \in \{\hat{\underline{+}}, \hat{\underline{-}}, \hat{\underline{*}}, \hat{\underline{/}}, \underline{\sqcap}\} \ : \ \hat{\mathbb{Z}} \times \hat{\mathbb{Z}} \to \hat{\mathbb{Z}}$$

$$A \hat{\diamond} B = |\bigcup \{a \hat{\underline{\diamond}} b \mid a \in A, b \in B\}|$$

구간 도메인과 마찬가지로 구간 집합 도메인에서도 분석을 유한시간에 끝내기 위한 넓히기(widening)연산과 정확도를 복구하는 좁히기(narrowing) 연산이 필요하다. Umirac에서는 구간 집합을 하나의 구간으로 요약한 후 구간 사이의 넓히기/좁히기 연산을 수행하도록 구간 집합 사이의 넓히기/좁히기 연산을 설계하였다. $\underline{x}$와 $\underline{y}$가 각각 구간 집합 x와 y를 하나의 구간으로 요약한 것이라 할 때 넓히기 연산자 ∇와 좁히기 연산자 Δ는 다음과 같이 정의된다.

– 넓히기(widening)

$$\begin{aligned}
x \nabla y &= \{\underline{x} \underline{\nabla} \underline{y}\} \\
\underline{\bot} \underline{\nabla} \underline{y} &= \underline{y} \\
\underline{x} \underline{\nabla} \underline{\bot} &= \underline{x} \\
[l_0, u_0] \underline{\nabla} [l_1, u_1] &= [0 \le l_1 < l_0 \ ? \ 0 : (l_1 < l_0 \ ? \ -\infty : l_0), \\
&\quad\ 0 \ge u_1 > u_0 \ ? \ 0 : (u_0 < u_1 \ ? \ +\infty : u_0)]
\end{aligned}$$

– 좁히기(narrowing)

$$\begin{aligned}
x \Delta y &= x \sqcap \{\underline{x} \underline{\Delta} \underline{y}\} \\
\underline{\bot} \underline{\Delta} \underline{y} &= \underline{\bot} \\
\underline{x} \underline{\Delta} \underline{\bot} &= \underline{\bot} \\
[l_0, u_0] \underline{\Delta} [l_1, u_1] &= [((l_0 \le 0 \le l_1) \vee (l_0 = -\infty)) \ ? \ l_1 : l_0, \\
&\quad\ ((u_1 \le 0 \le u_0) \vee (u_0 = +\infty)) \ ? \ u_1 : u_0]
\end{aligned}$$

사용되지 않는 변수와 구조체 필드 분석을 위해, G의 요약공간을 어떤 메모리가 사용될 수 있는지의 여부를 표현하도록 확장하였다:

$$\hat{Map} = \hat{Addr} \xrightarrow{\text{fin}} (\hat{Value} \times \mu)$$

$\hat{Map}$은 메모리의 요약을 의미하며, μ는 $\bot$(사용되지 않음)과 $\top$(사용될 수 있음)의 두 가지 상태를 표현한다. 두 요약된 메모리를 합칠 때, 같은 주소에 대해 서로 μ값이 다르다면 합친 메모리의 μ값은 $\top$으로 정의한다. 그리고 요약 의미 공간에서 값을 계산하는 함수 $\hat{\mathcal{V}}$를 다음과 같이 정의한다:

$$\hat{\mathcal{V}} : \hat{Map} \to \hat{Expr} \to (\hat{Value} \times \hat{Map})$$

Umirac의 $\hat{\mathcal{V}}$는 메모리와 수식(expression)을 받아 계산한 값과 메모리를 내놓는다. 메모리를 반환하는 이유는 $\hat{\mathcal{V}}$는 메모리에서 값을 꺼냈을 경우 해당 메모리 공간에 있는 μ값이 $\bot$이었으면 $\top$으로 바꾸는데, 이 때 바뀐 메모리를 물려주기 위함이다.

Umirac은 고정점을 구한 분석결과를 가지고 프로그램의 실행 흐름을 따라가며 반드시 사용되지 않을 자료만을 찾는다. 요약해석의 틀에서 설계된 Umirac은 어떤 메모리 공간이 실제 실행 중에 사용될 수 있는지의 여부와 버퍼의 인덱스 값으로 사용될 수 있는 값들을 모두 포섭한 분석을 수행한다. 따라서 이 분석결과를 마탕으로 Umirac이 사용되지 않는 자료라 판단한 것은 항상 실제로 사용되지 않는 자료들이라 할 수 있다.

하지만 분석 대상 프로그램의 실행의미가 충분하지 않다면 정확하지 않은 경보를 발생시킬 수 있다. 예를 들어 변수나 버퍼를 사용하는 부분이 다른 언어로 기술되어 있거나, 외부함수로 정의되어 있어 분석 대상 프로그램에 나타나 있지 않은 경우 Umirac은 해당하는 변수나 구조체 필드, 버퍼 구간을 사용하지 않는다고 분석한다.

3 실험 결과

실제 임베디드 시스템 소프트웨어와 리눅스 커널 코드를 대상으로 Umirac의 성능을 측정하였다. 수행속도의 대조군으로 Airac5를 더욱 다듬은 Airac7[6]을 설정하였다. 표 I, II는 각각 임베디드 시스템 소프트웨어와 리눅스 커널 코드의 일부를 대상으로 한 Umirac의 실험결과와 Airac7와의 수행 속도 비교를 보여준다.

실험은 모두 인텔 펜티엄4 3.2GHz 프로세서와 4GB 메모리가 탑재된 리눅스 시스템에서 수행하였다. Umirac과 Airac7 모두 분석 시나리오를 만들지 않고 프로그램의 모든 함수를 차례대로 분석하도록 하였으며, Airac7의 정확도 옵션은 가장 세밀한 분석을 수행하는 3으로 지정하였다. 함수 호출 펼치기(function inlining)는 두 분석기 모두 한 차례 하는 것으로, 그리고 분석 시간은 전처리와 변환을 제외한 분석에만 소요된 CPU 시간으로 측정하였다. 또한 프로그램의 크기는 프로그램을 전처리 하기 전의 것을 기록하였다.

프로그램 (LOC)	Umirac alarm			Umirac	Airac7
	UV	UF	BB	시간(sec)	시간(sec)
AviReader (1486)	5(0)	0(0)	6(3)	6	4
H263FRDivx (3944)	7(0)	4(0)	5(0)	300	257
MADStream (8171)	17(1)	0(0)	31(7)	3615	2681
software1 (4725)	7(0)	1(0)	8(0)	344	291
software2 (21653)	16(2)	2(0)	30(10)	6510	3948

[표 I]임베디드 시스템 소프트웨어

프로그램 (LOC)	Umirac alarm			Umirac	Airac7
	UV	UF	BB	시간(sec)	시간(sec)
vmax301.c (246)	31(8)	2(2)	1(1)	101	82
cdc_acm.c (849)	48(12)	4(3)	4(3)	187	139
atkbd.c (944)	40(7)	3(3)	3(2)	333	268
eata_pio.c (984)	105(21)	6(5)	11(6)	1120	803
ip6_output.c (1110)	140(29)	2(2)	24(13)	6430	4773
xfrm_user.c (1201)	148(35)	3(1)	19(9)	5199	3794
keyboard.c(1256)	105(26)	0(0)	17(10)	443	359
af_inet.c (1273)	150(38)	3(3)	15(10)	6590	4917
usb_midi.c (2206)	104(17)	2(1)	13(5)	2301	1619
aty128fb.c (2466)	104(25)	1(1)	14(10)	680	525
mptbase.c (6158)	150(53)	3(2)	27(20)	10817	8804

[표 II]Linux Kernel 2.6.4의 일부

Umirac의 분석 결과에서 UV, UF, BB는 각각 사용하지 않는 변수, 구조체의 필드, 그리고 버퍼 구간의 개수를 나타내며, 괄호 안의 숫자는 외부 함수 또는 다른 언어로 구현된 부분에서 사용되어지는 것으로 보이는 부정확한 경보의 수이다. 부정확한 경보의 수는 표 II에서 더욱 많이 관찰되는데, 이는 프로그램의 특성에 기인한 것으로, 리눅스 커널 프로그램은

많은 시스템 헤더 파일을 사용하여 감추어진 외부함수가 많고 어셈블리어 등 다른 언어로 기술된 부분이 많아 정확하지 않은 경보가 최대 50% 가까이 발생하는 것을 볼 수 있다. 한편 표 I의 임베디드 시스템 소프트웨어의 경우 다른 언어로 구현된 부분이 없고 외부함수 호출이 적어 부정확한 경보가 전체 경보의 0 – 32% 정도로 나타났다.

Airac7와의 수행 속도 비교에서는 Umirac이 60 – 86% 정도의 속도를 보여준다는 것을 알 수 있다. 확장된 요약 도메인과 요약 실행 공간이 분석의 비용을 증가시킨 것으로 보인다.

4 결론 및 앞으로

Umirac은 Airac5의 틀을 바탕으로 하여 만들어진, C 프로그램에서 사용되지 않는 변수, 구조체의 필드, 그리고 버퍼의 구간을 찾아내는 요약해석기이다. 이를 위해 구간 집합 도메인을 사용한 요약해석을 설계하고 구현하였다.

요약해석의 틀에서 설계된 Umirac은 반드시 사용되지 않을 자료들만을 찾아내지만 프로그램의 모든 실행의미가 드러나있지 않은 경우 정확하지 않은 경보를 발생시킬 수 있다.

리눅스 커널과 임베디드 시스템 소프트웨어와 같은 실제 프로그램에 대한 실험을 통해 사용되지 않는 자료들을 찾음으로써 요약해석이 메모리 사용량 최적화에 유용한 정보를 제공할 수 있다는 것을 보였다. Umirac의 분석을 위해 확장한 설계는 기존의 구간 분석 요약해석에 비해 60 – 86% 정도의 수행속도를 보였다.

앞으로의 연구는 분석기가 부정확한 분석을 하는 경우를 줄이는 방법과 분석 결과를 활용하는 방법에 대한 것이다. 먼저 분석기의 정확도를 높이기 위해 자주 사용되는 외부 함수들의 의미를 Umirac에 맞게 정의하여 분석기에 탑재하는 과정을 구현하는 중이다. 또한 Umirac의 분석 정보를 활용하여 프로그램에서 사용되지 않는 자료를 자동으로 없애는 프로그램 변환 규칙을 정의하는 연구를 진행하고 있다. 예를 들어 어떤 버퍼에서 사용하지 않는 구간이 발견되었다면, 사용하지 않는 구간이 없게끔 버퍼의 크기를 줄이고 프로그램의 의미가 변하지 않게 버퍼를 접근하는 수식들을 바꾸어주는 것을 생각할 수 있다.

감사의 글

본 연구에 있어 Umirac의 초기 설계 및 구현에 도움을 준 오학주에게 감사의 뜻을 전한다.

참고문헌

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[2] Patrick Cousot. Abstract interpretation. *Symposium on Models of Programming Languages and Computation, ACM*, 28(2):324–328, June 1996.

[3] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 238–252, January 1977.

[4] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 269–282, 1979.

[5] B. W. Kernighan and D. M. Ritchie. *The C programming language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1978.

[6] http://ropas.snu.ac.kr/airac7.

[7] Jaeho Shin. An abstract interpretation with the interval domain for C-like programs. Master's thesis, Seoul National University, 2006.

[8] Jaeho Shin, Jaehwang Kim, Hakjoo Oh, Yungbum Jung, and Kwangkeun Yi. Abstact interpreter AiracV. In *Transactions on Programming Languages*, volume 19, pages 11–17. Korea Information Science Society Special Interest Group on Programming Languages, Nov 2005.

[9] Misha Zitser, Richard Lippmann, and Tim Leek. Testing static analysis tools using exploitable buffer overflows from open source code. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 97–106. ACM Press, 2004.

황 의 권

– 2001-2005 연세대학교 학사
– 2005-현재 서울대학교 석사과정

<관심분야> 프로그램 분석 및 검증

이 광 근

– 1983-1987 서울대학교 학사
– 1988-1990 Univ. of Illinois at Urbana-Champaign 석사
– 1990-1993 Univ. of Illinois at Urbana-Champaign 박사
– 1993-1995 Bell Labs., Murray Hill 연구원
– 1995-2003 한국과학기술원 교수
– 2003-현재 서울대학교 교수

<관심분야> 프로그램 분석 및 검증, 프로그래밍 언어