(19) OFICINA ESPAÑOLA DE PATENTES Y MARCAS

ESPAÑA

(11) Número de publicación: **2 733 516**

(51) Int. CI.:

G06F 8/41 (2008.01)

(12) TRADUCCIÓN DE PATENTE EUROPEA T3

(86) Fecha de presentación y número de la solicitud internacional: **09.09.2011 PCT/US2011/051023**

(87) Fecha y número de publicación internacional: **12.04.2012 WO12047447**

(96) Fecha de presentación y número de la solicitud europea: **09.09.2011 E 11831155 (4)**

(97) Fecha y número de publicación de la concesión europea: **10.04.2019 EP 2622466**

(54) Título: **Verificación de límites en tiempo de compilación para tipos definidos por el usuario**

(30) Prioridad:

28.09.2010 US 892291

(45) Fecha de publicación y mención en BOPI de la traducción de la patente:
29.11.2019

(73) Titular/es:

MICROSOFT TECHNOLOGY LICENSING, LLC (100.0%)
One Microsoft Way
Redmond, WA 98052, US

(72) Inventor/es:

HARVEY, DANIEL STEPHEN

(74) Agente/Representante:

CARPINTERO LÓPEZ, Mario

DESCRIPCIÓN

Verificación de límites en tiempo de compilación para tipos definidos por el usuario

Antecedentes

La verificación de límites se puede realizar en programas de computación para detectar si una variable está dentro de ciertos límites especificados. Por ejemplo, un valor se puede verificar antes de que se use como un índice en una matriz para determinar si el valor se encuentra dentro de los límites de la matriz; este tipo de verificación de límites a veces se denomina verificación de índice o verificación de rango. Una verificación de límites fallida puede generar un error de tiempo de ejecución, como una señal de excepción. Un intento de acceso a una matriz u otra variable fuera de sus límites a menudo indica un error de programación. Sin embargo, no siempre se realiza una verificación de límites antes de cada uso de una variable limitada, ya que las verificaciones de límites aumentan el tiempo de ejecución del programa.

Los compiladores a veces eliminan automáticamente las verificaciones de límites que se consideran innecesarias. Como ejemplo, considere el código de programa que lee un valor de una ubicación dentro de una matriz y luego almacena otro valor (o el mismo valor) en esa misma ubicación. Sin ninguna optimización, este código podría incluir una primera verificación de límites cuando la ubicación de la matriz se lee desde la matriz y una segunda verificación de límites cuando se escribe la misma ubicación en la matriz. Pero un compilador u otra herramienta de optimización puede eliminar automáticamente la segunda verificación de límites después de determinar que el tamaño de la matriz no se modifica y que la misma ubicación en la matriz se está leyendo y luego se escribe. De manera más general, se utilizan diversas optimizaciones en los compiladores y otras herramientas para reducir o eliminar automáticamente las verificaciones de límites redundantes. Del documento "Safe bounds check annotations" (Anotaciones de verificación de límites seguros) de Jeffrey von Ronne *et. al.*, se conoce un armazón en el que se pueden usar de forma segura análisis de rango estático durante la compilación de fuente llama de rendimiento no crítico en la capa Moabite independiente de la máquina. Allí, los resultados de los análisis estáticos se utilizan para obtener pruebas de que se mantienen ciertas restricciones de desigualdad lineal. De este modo, las verificaciones de límites de matriz se pueden eliminar durante la compilación sin un análisis de tiempo de ejecución costoso.

Además, a partir del documento "Pluggable type-checking for custom type qualifiers in JAVA" (Verificación de tipo conectable para calificadores de tipo personalizado en JAVA) por Matthew M. Papi et al., se conoce otro armazón para agregar calificadores de tipo personalizados al lenguaje JAVA de una manera compatible con versiones anteriores. Los calificadores son definidos por el diseñador del sistema y se crea un complemento del compilador que impone la semántica respectiva.

Además, el documento "Memory management based on method invocation in RTSJ" (Gestión de memoria basada en invocación de procedimiento en RTSJ) por Jagun Kwon et al. describe un modelo de gestión de memoria para el perfil Ravenscar-JAVA. El enfoque desvelado asigna una y la mayor parte del área de memoria al procedimiento que puede especificar el usuario por medio de los tipos de anotación de JAVA 1.5. De este modo, se eliminan las verificaciones de las reglas principales y se simplifican otras verificaciones en tiempo de ejecución que son la causa principal de imprevisibilidad y sobrecargos.

Sumario

La verificación automatizada de límites en accesos de matriz no se usa siempre, debido a su impacto real (o percibido) en el rendimiento del programa. Sin embargo, la verificación automática de límites está disponible para los desarrolladores que optan por usarla para un acceso más seguro a las matrices. Sin embargo, cuando los desarrolladores usan tipos de datos personalizados que tienen una estructura más intrincada, o se apartan de una matriz simple, la verificación de violaciones de acceso a veces ha implicado la inserción manual de código de verificación de límites. Desafortunadamente, cuando los desarrolladores escriben verificaciones de límites explícitamente, el propósito del código de verificación de límites puede no ser evidente para un compilador, por lo que el compilador carece de conocimientos que le permitan eliminar las verificaciones de límites redundantes.

Algunas realizaciones descritas en la presente memoria proporcionan la verificación automática de límites en tiempo de compilación de tipos definidos por el usuario, en parte identificando en un código fuente una clase definida por el usuario para acceder de manera segura a la memoria asignada explícitamente. La clase definida por el usuario tiene un miembro de código de acceso a la memoria que fue anotado por un desarrollador con una anotación de verificación de límites definidos por el usuario, por ejemplo, una anotación que indica al compilador que el código accede a un búfer asignado en la memoria u otra memoria asignada explícitamente. La clase definida por el usuario también tiene un miembro proveedor de límites que está anotado por un desarrollador para indicar al compilador que proporciona información de límites para generar una verificación de límites en el código de acceso a la memoria. El miembro que proporciona el límite puede ser un campo de número entero que contiene un límite, por ejemplo, o un procedimiento que devuelve un límite cuando se le llama.

La verificación de límites definidos por el usuario puede proporcionar una verificación de límites donde el lenguaje de programación no tiene ninguno, o puede complementar las verificaciones de límites existentes, por ejemplo, envolviendo un tipo de matriz integrado o un tipo administrado por recolector de residuos. La verificación de límites se puede extender más allá de las matrices y otros tipos cuyo diseño es controlado por un compilador; la clase definida por el usuario puede definirse sin utilizar ningún tipo de matriz de elementos múltiples como un tipo constituyente.

El compilador inserta una representación de verificación de límites de la anotación de verificación de límites definidos por el usuario en un código de lenguaje intermedio. Luego, una optimización reduce automáticamente la verificación de límites duplicada que de lo contrario se produciría en el código ejecutable. La optimización se puede aplicar a las representaciones de verificación de límites del lenguaje intermedio, al código de verificación de límites insertado, o a ambos.

Los ejemplos dados son meramente ilustrativos. Este sumario no tiene la intención de identificar factores clave o características esenciales del objeto reivindicado, ni está destinado a ser utilizado para limitar el alcance del tema reivindicado. Por el contrario, este sumario se proporciona para presentar, en forma simplificada, algunos conceptos que se describen con más detalle a continuación en la Descripción detallada. La invención se define en las reivindicaciones adjuntas.

Breve descripción de las figuras

Se dará una descripción más particular con referencia a las figuras adjuntas. Estas figuras solo ilustran aspectos seleccionados y, por lo tanto, no determinan completamente la cobertura o el alcance.

La Figura 1 es un diagrama de bloques que ilustra un sistema de computación que tiene al menos un procesador, al menos una memoria, al menos un código fuente de programa y otros elementos en un entorno operativo que pueden estar presentes en varios nodos de red, y también ilustran realizaciones de medio de almacenamiento configurado.

La Figura 2 es un diagrama de bloques que ilustra la verificación de límites en tiempo de compilación de tipos de datos definidos por el usuario arbitrariamente complejos, en una arquitectura a modo de ejemplo.

La Figura 3 es un diagrama de flujo que ilustra las etapas de algunos procesos y realizaciones de medio de almacenamiento configurado.

Descripción detallada

Descripción general

Los sistemas de códigos administrados actuales, como los de los entornos Microsoft® .NET y Java® (marcas de Microsoft Corporation y Oracle America, Inc., respectivamente), a menudo suponen que toda la memoria utilizada por el programa de códigos administrados está asignada y administrada automáticamente. Sin embargo, en la programación de sistemas, y particularmente en la programación de controladores de dispositivos, esta suposición puede fallar. En algunos casos, la memoria utilizada por un controlador de dispositivo está fija en su posición por el dispositivo físico, como cuando un búfer en el dispositivo se asigna en memoria a un conjunto específico de direcciones. En otros casos, la memoria se gestiona explícitamente para evitar la copia innecesaria de datos o para reutilizar los búferes dentro de un cierto umbral de tiempo.

En algunos lenguajes, como C#, cuando un programa utiliza memoria asignada explícitamente, un programador utilizará con frecuencia un puntero no seguro a la estructura de datos no administrada. Se debe tener mucho cuidado para evitar errores al acceder al puntero o al hacer aritmética de punteros. Los punteros no seguros pueden anular los beneficios de corrección del código administrado, ya que una aritmética de punteros incorrecta podría provocar daños en la memoria o un fallo del programa. Tales errores pueden ser particularmente difíciles de depurar en presencia de recolección de residuos. Cuando los programadores escriben estructuras de datos personalizadas y desean realizar verificaciones de límites para seguridad, pueden escribir explícitamente las verificaciones de límites como líneas de código fuente. El compilador no puede distinguir el propósito de dicho código del propósito de otras líneas del código, en cuyo caso el compilador carece de información que le permita eliminar las verificaciones de límites explícitas.

Algunas realizaciones descritas aquí permiten que el código administrado acceda de manera segura y eficiente a la memoria asignada explícitamente. Por lo tanto, el código administrado se puede utilizar de manera eficiente para la programación de sistemas. Al usar código administrado para la programación de sistemas con verificación de límites optimizados en tiempo de compilación como se describe en la presente memoria, los programadores pueden escribir controladores de dispositivo eficientes en código administrado y eliminar errores comunes en los controladores de dispositivo que son causas frecuentes de fallas de sistema operativo.

Algunas realizaciones descritas en la presente memoria pueden verse en un contexto más amplio. Por ejemplo, conceptos como el acceso a la memoria, los límites de las variables, la anotación del código fuente y la compilación, pueden ser relevantes para una realización particular. Sin embargo, de la disponibilidad de un contexto amplio no se deduce que se busquen derechos exclusivos en la presente memoria para obtener ideas abstractas; esto no es así. Por el contrario, la presente divulgación se centra en proporcionar realizaciones específicas apropiadas. Otros medios, sistemas y procedimientos que involucran acceso a la memoria, verificación de límites, compilación y/o anotación de código fuente, por ejemplo, están fuera del alcance actual. En consecuencia, la vaguedad y los problemas de prueba que lo acompañan también se evitan bajo una comprensión adecuada de la presente divulgación.

Ahora se hará referencia a realizaciones ejemplares, tales como las ilustradas en las figuras, y se usará un lenguaje específico en la presente memoria para describir las mismas. Pero las modificaciones y otras modificaciones de las características ilustradas en la presente memoria, y las aplicaciones adicionales de los principios ilustrados en la presente memoria, que se producirían por un experto en la(s) técnica(s) relevante(s) y que posean esta divulgación, deben considerarse dentro del alcance de las reivindicaciones.

El significado de los términos se aclara en esta divulgación, por lo que las reivindicaciones deben leerse prestando especial atención a estas aclaraciones. Se proporcionan ejemplos específicos, pero los expertos en la(s) técnica(s) relevante(s) entenderán que otros ejemplos también pueden caer dentro del significado de los términos utilizados, y dentro del alcance de una o más reivindicaciones. Los términos no necesariamente tienen el mismo significado aquí que tienen en el uso general, en el uso de una industria en particular, o en un diccionario o conjunto de diccionarios en particular. Los números de referencia se pueden usar con varias frases para ayudar a mostrar la amplitud de un término. La omisión de un número de referencia de un fragmento de texto dado no significa necesariamente que el contenido de una Figura no esté siendo discutido por el texto. El inventor afirma y ejerce su derecho a su propia lexicografía. Los términos pueden definirse, explícita o implícitamente, aquí en la Descripción detallada y/o en cualquier otra parte del archivo de la solicitud.

Como se usa en la presente memoria, un "sistema de computación" puede incluir, por ejemplo, uno o más servidores, placas base, nodos de procesamiento, ordenadores personales (portátiles o no), asistentes digitales personales, teléfonos celulares o móviles y/o dispositivo(s) que proporciona(n) uno o más procesadores controlados al menos en parte por instrucciones. Las instrucciones pueden estar en forma de software en memoria y/o circuitos especializados. En particular, aunque puede ocurrir que muchas realizaciones se ejecuten en estaciones de trabajo u ordenadores portátiles, otras realizaciones pueden ejecutarse en otros dispositivos de computación, y cualquiera o más de dichos dispositivos pueden ser parte de una realización dada.

Un sistema de computación "multihilo" es un sistema de computación que soporta múltiples hilos de ejecución. Se debe entender que el término "hilo" incluye cualquier código que pueda o esté sujeto a sincronización, y también puede ser conocido por otro nombre, como "tarea", "proceso" o "corutina", por ejemplo. Los hilos pueden ejecutarse en paralelo, en secuencia o en una combinación de ejecución paralela (por ejemplo, mediante multiprocesamiento) y ejecución secuencial (por ejemplo, en intervalos de tiempo). Los entornos multihilo han sido diseñados en varias configuraciones. Los hilos de ejecución pueden ejecutarse en paralelo, o los hilos pueden organizarse para su ejecución en paralelo, pero en realidad se turnan para ejecutarse en secuencia. El sistema multihilo se puede implementar, por ejemplo, ejecutando diferentes hilos en diferentes núcleos en un entorno de multiprocesamiento, dividiendo en el tiempo diferentes hilos en un solo núcleo del procesador, o mediante una combinación de hilado en intervalos de tiempo y mediante multiprocesador. Los cambios de contexto de hilos pueden iniciarse, por ejemplo, mediante un programador de hilos del núcleo, mediante señales en el espacio de usuario o mediante una combinación de operaciones de núcleo y de espacio de usuario. Los hilos pueden turnarse para operar con datos compartidos, o cada hilo puede operar con sus propios datos, por ejemplo.

Un "procesador lógico" o "procesador" es una unidad única de procesamiento de hilos de hardware independiente. Por ejemplo, un chip de cuatro núcleos hiperhilado que ejecuta dos hilos por núcleo tiene ocho procesadores lógicos. Los procesadores pueden ser de uso general o pueden adaptarse a usos específicos, como el procesamiento de gráficos, el procesamiento de señales, el procesamiento aritmético de punto flotante, el cifrado, el procesamiento de E/S, etc.

Un sistema de computación "multiprocesador" es un sistema de computación que tiene múltiples procesadores lógicos. Los entornos multiprocesador se encuentran en varias configuraciones. En una configuración dada, todos los procesadores pueden ser funcionalmente iguales, mientras que, en otra configuración, algunos procesadores pueden diferir de otros procesadores en virtud de tener diferentes capacidades de hardware, diferentes asignaciones de software, o ambas. Dependiendo de la configuración, los procesadores pueden estar estrechamente acoplados entre sí en un solo bus, o pueden estar débilmente acoplados. En algunas configuraciones, los procesadores comparten una memoria central, en algunos tienen cada uno su propia memoria local y en algunas configuraciones, tanto las memorias compartidas como las locales están presentes.

Los "núcleos" incluyen sistemas operativos, hipervisores, máquinas virtuales y software de interfaz de hardware

similar.

"Código" significa instrucciones, datos del procesador (que incluyen constantes, variables y estructuras de datos), o tanto instrucciones como datos.

El "programa" se usa ampliamente aquí, para incluir aplicaciones, núcleos, controladores, controladores de interrupciones, bibliotecas y otros códigos escritos por programadores (a los que también se hace referencia como desarrolladores).

"Automáticamente" significa mediante el uso de automatización (por ejemplo, hardware de computación de propósito general configurado por software para operaciones específicas discutidas en la presente memoria), a diferencia de sin automatización. En particular, las etapas realizadas "automáticamente" no se realizan a mano en papel o en la mente de una persona; se realizan con una máquina. Sin embargo, "automáticamente" no significa necesariamente "inmediatamente".

A lo largo de este documento, el uso del plural opcional "(s)" o "(es)" significa que una o más de las características indicadas están presentes. Por ejemplo, "anotación(es)" significa "una o más anotaciones" o equivalentemente "al menos una anotación".

En todo este documento, a menos que se indique expresamente lo contrario, cualquier referencia a una etapa en un proceso supone que la etapa puede ser realizada directamente por una parte interesada y/o realizada indirectamente por la parte a través de mecanismos y/o entidades que intervienen, y que todavía se encuentran dentro del alcance de la etapa. Es decir, no se requiere la ejecución directa de la etapa por la parte interesada a menos que la ejecución directa sea un requisito expresamente establecido. Por ejemplo, una etapa que involucra la acción de una parte interesada como "transmitir", "enviar", "comunicar", "aplicar", "insertar", anotar "," denotar ", "especificar ", o apuntar de otro modo a un destino puede implicar una acción de intervención, como reenviar, copiar, cargar, descargar, codificar, descodificar, comprimir, descomprimir, cifrar, descifrar, etc., por alguna otra parte, pero aun así debe entenderse como realizada directamente por la parte interesada.

Cuando se hace referencia a datos o instrucciones, se entiende que estos elementos configuran una memoria legible por ordenador transformándola en un artículo en particular, en lugar de simplemente existir en un papel, en la mente de una persona, o como una señal transitoria en un cable, por ejemplo.

Entornos operativos

Con referencia a la Figura 1, un entorno operativo 100 para una realización puede incluir un sistema de computación 102.El sistema de computación 102 puede ser un sistema de computación multiprocesador, o no. Un entorno operativo puede incluir una o más máquinas en un sistema de computación determinado, que puede estar en clúster, en la red cliente-servidor y/o en una red entre pares.

Los usuarios humanos 104 pueden interactuar con el sistema de computación 102 utilizando pantallas, teclados y otros periféricos 106.Los administradores de sistemas, desarrolladores, ingenieros y usuarios finales son cada uno un tipo particular de usuario 104. Los agentes automatizados que actúan en nombre de una o más personas también pueden ser usuarios 104. Los dispositivos de almacenamiento y/o dispositivos de red pueden considerarse equipos periféricos en algunas realizaciones. Otros sistemas de computación que no se muestran en la Figura 1 pueden interactuar con el sistema de computación 102 o con otra realización del sistema utilizando una o más conexiones a una red 108 a través de un equipo de interfaz de red, por ejemplo.

El sistema de computación 102 incluye al menos un procesador lógico 110. El sistema de computación 102, al igual que otros sistemas adecuados, también incluye uno o más medios de almacenamiento no transitorios legibles por ordenador 112. Los medios 112 pueden ser de diferentes tipos físicos. Los medios 112 pueden ser memoria volátil, memoria no volátil, medios fijos, medios extraíbles, medios magnéticos, medios ópticos y/o de otros tipos de medios no transitorios (a diferencia de los medios transitorios, como un cable que simplemente propaga una señal). En particular, un medio configurado 114 como un CD, DVD, una tarjeta de memoria u otro medio de memoria no volátil extraíble puede convertirse en parte funcional del sistema de computación cuando se inserta o instala de otro modo, haciendo que su contenido sea accesible para que lo utilice el procesador 110. El medio configurado removible 114 es un ejemplo de un medio de almacenamiento 112 legible por ordenador. Algunos otros ejemplos de medios de almacenamiento 112 legibles por ordenador incluyen RAM, ROM, discos duros y otros dispositivos de almacenamiento integrados que los usuarios 104 no pueden eliminar fácilmente.

El medio 114 está configurado con instrucciones 116 que son ejecutables por un procesador 110; "ejecutable" se usa en un sentido amplio aquí para incluir código de máquina, código interpretable y código que se ejecuta en una máquina virtual, por ejemplo. El medio 114 también se configura con los datos 118 que se crean, modifican, hacen referencia y/o utilizan de otro modo mediante la ejecución de las instrucciones 116. Las instrucciones 116 y los datos 118 configuran el medio 114 en el que residen; cuando esa memoria es una parte funcional de un sistema

de computación dado, las instrucciones 116 y los datos 118 también configuran ese sistema de computación. En algunas realizaciones, una parte de los datos 118 es representativa de elementos del mundo real tales como características de producto, inventarios, medidas físicas, configuraciones, imágenes, lecturas, objetivos, volúmenes, etc. Dichos datos también se transforman mediante la verificación flexible de límites optimizados en tiempo de compilación, como se explica en la presente memoria, por ejemplo, insertando, aplicando, especificando, anotando, denotando, vinculando, implementando, modificando, visualizando, creando, cargando y/u otras operaciones.

Un programa 120 (con código fuente 122, código de lenguaje intermedio 124 y código ejecutable 126, por ejemplo), depuradores, compiladores y otras herramientas de desarrollo 136, otro software y otros elementos mostrados en las Figuras pueden residir parcial o totalmente dentro de uno o más medios 112, configurando de esta forma esos medios. El código de lenguaje intermedio 124 a veces se conoce como una representación intermedia. El programa 120 puede incluir tipos integrados 128 y tipos gestionados por el recolector de residuos 130, por ejemplo. En muchas configuraciones de desarrollo, los tipos de matrices 132 están integrados y administrados. Además del (los) procesador(es) 110, un entorno operativo puede incluir otro hardware, como pantallas, dispositivos mapeados en memoria 134, buses, fuentes de alimentación y aceleradores, por ejemplo.

Un determinado entorno operativo 100 puede incluir un Entorno de Desarrollo Integrado (IDE) 138 que proporciona a un desarrollador un conjunto de herramientas de desarrollo de software coordinadas. En particular, algunos de los entornos operativos adecuados para algunas realizaciones incluyen o ayudan a crear un entorno de desarrollo Microsoft® Visual Studio® (marcas de Microsoft Corporation) configurado para admitir el desarrollo de programas. Algunos entornos operativos adecuados incluyen entornos Java® (marca de Oracle America, Inc.), y algunos incluyen entornos que utilizan lenguajes como C++ o C# ("C-Sharp"), pero las enseñanzas que se incluyen en la presente memoria son aplicables para una amplia variedad de lenguajes de programación, modelos de programación y programas.

Uno o más elementos se muestran en forma de esquema en la Figura 1 para enfatizar que no son necesariamente parte del entorno operativo ilustrado, pero pueden interoperar con elementos en el entorno operativo como se describe en la presente memoria. No se sigue que los elementos que no están en forma de esquema son necesariamente requeridos, en cualquier Figura o en cualquier realización.

Sistemas

La Figura 2 ilustra una arquitectura que es adecuada para su uso con algunas realizaciones. Un tipo definido por el usuario 204, tal como una clase definida por el usuario 202, tiene anotaciones 206 para transmitir las intenciones de verificación de límites del desarrollador a un compilador 224 de una manera que le permite al compilador no solamente proporcionar la verificación de límites sino también eliminar las verificaciones de límites redundantes. Las anotaciones pueden identificar, por ejemplo, el código de acceso a la memoria 208 y el código que proporciona límites 210 en el tipo definido por el usuario. El código de acceso a la memoria puede ser declaraciones en línea y/o procedimientos distintos, por ejemplo. El código de provisión de límites puede ser campos 212 que contengan límites 218 y/o procedimientos 214 que devuelvan los límites 218 cuando son llamados.

Aunque la verificación de índice de matriz puede ser muy útil, el enfoque en la presente memoria está en otros tipos de verificación de límites, a saber, verificación de límites para estructuras definidas por el usuario que no son meras matrices. A diferencia de un tipo de matriz familiar 132, la clase 202 u otro tipo definido por el usuario no está integrado y, por lo tanto, puede tener un diseño de datos 216 que no está controlado por el compilador 224.

En algunas realizaciones, el compilador 224 inserta representaciones de verificación de límites 220 en el código de lenguaje intermedio 124 en respuesta a las anotaciones 206.El código de verificación de límites 222 se coloca posteriormente en el código ejecutable 126 en respuesta a las representaciones de verificación de límites 220. Las representaciones de verificación de límites 220 pueden respetar las convenciones familiares para el código de lenguaje intermedio 124, y el código de verificación de límites 222 generado puede incluir instrucciones de salto condicional conocidas y similares. Sin embargo, el contexto de estas convenciones e instrucciones familiares, para los fines actuales, es de tipos definidos por el usuario que no son simples matrices y que en algunas realizaciones ni siquiera utilizan matrices como tipos constituyentes.

En algunas realizaciones, un optimizador 226 aplica la(s) optimización(es) 228 a las representaciones de verificación de límites 220, al código de verificación de límites 222, o ambos, para eliminar la verificación de límites redundante que de lo contrario ocurriría en el código ejecutable 126. El optimizador 226 puede estar integrado en el compilador 224, o puede ser una herramienta separada que es invocada por el compilador 224 o por el desarrollador, dependiendo de la realización. Las optimizaciones que se utilizan con la verificación de límites de matriz pueden adaptarse y aplicarse a los tipos definidos por el usuario. Dado el beneficio de las anotaciones 206 en un tipo 204, por ejemplo, el optimizador puede determinar que todos los accesos dentro de un bucle a una variable de ese tipo definido por el usuario 204 están dentro de los límites de dirección de memoria permitidos de la variable y, por lo tanto, el optimizador puede eliminar múltiples verificaciones de límites que de lo contrario se

producirían como resultado de la ejecución del bucle.

Como lo sugiere la Figura 2, los tipos definidos por el usuario y las verificaciones de límites optimizadas en tiempo de compilación pueden ser particularmente útiles en el desarrollo del código del controlador de dispositivo 230 como el programa 120. El código administrado se puede usar para la programación de sistemas, y el controlador del dispositivo se puede desarrollar utilizando un IDE 138 para su ejecución en un sistema que proporciona una recolección de residuos de memoria que no está asignada explícitamente. La memoria que se asigna explícitamente y, por lo tanto, no se recolecta como residuos, puede ser administrada por el código del desarrollador sin sacrificar la verificación de límites y sin imponer una verificación de límites extremadamente ineficiente. Por ejemplo, una clase 202 puede definirse para incluir un búfer asignado explícitamente mapeado en memoria 232 para un dispositivo 134, con un búfer anotado que accede 208 al procedimiento o procedimientos para leer/escribir el búfer. El tamaño del búfer puede determinarse dinámicamente y luego proporcionarse al código de verificación de límites por medio de una anotación 206 y un mecanismo de provisión de límite 210, como un campo bufferBound 212 o un procedimiento getBufferBound() 214.

Con referencia a las Figuras 1 y 2, algunas realizaciones proporcionan un sistema de computación 102 con un procesador lógico 110 y un medio de memoria 112 configurado por circuitos, firmware y/o software para transformar los códigos 122, 124, 126 en apoyo de la verificación de límites en tiempo de compilación optimizada como se describe en la presente memoria. La memoria está en comunicación operable con el procesador lógico. Un código fuente 122 que reside en la memoria tiene un tipo definido por el usuario 204. El tipo definido por el usuario tiene un procedimiento de acceso a la memoria 208 que se anota con una anotación de verificación de límites definidos por el usuario 206. El tipo definido por el usuario también tiene al menos un especificador de límite, como un campo o procedimiento que proporciona límites 210. Un compilador 224 que reside en la memoria está configurado para insertar en un código de lenguaje intermedio 124 una representación de verificación de límites 220 de la anotación de verificación de límites definidos por el usuario. Un optimizador 226 que reside en la memoria está configurado para aplicar una optimización 228 al código de lenguaje intermedio con el fin de reducir la verificación de límites duplicada.

En algunas realizaciones, el código fuente anotado incluye el código fuente 122 del controlador de dispositivo 230, y el tipo definido por el usuario 204 corresponde a un búfer mapeado en memoria 232.Los búferes mapeados en memoria son meramente un ejemplo; en algunas realizaciones, el código del controlador del dispositivo anotado accede a otra memoria asignada explícitamente 112.

En algunas realizaciones, el código fuente anotado incluye tipos de datos recolectados de residuos 130, y el tipo definido por el usuario corresponde a la memoria asignada explícitamente. En algunas realizaciones, el tipo definido por el usuario 204 tiene un diseño de datos 216 que no está controlado por el compilador 224. En algunas realizaciones, el tipo definido por el usuario 204 se define como libre de cualquier tipo de matriz de elementos múltiples como un tipo constituyente. En otras, el tipo definido por el usuario 204 tiene una o más matrices como tipos constituyentes, pero es más complejo que una matriz. En algunas, el tipo definido por el usuario 204 envuelve un tipo de matriz y proporciona una verificación de límites complementaria, por ejemplo, para verificar no solamente que un acceso a la matriz no solo se encuentra dentro del espacio asignado a la matriz, sino que también se encuentra dentro del espacio que contiene elementos actualizados, o dentro de una subporción de una matriz que el desarrollador quiere que mantenga un conjunto específico de valores, por ejemplo.

En algunas realizaciones, el especificador de límite (también conocido como mecanismo de provisión de límite 210) incluye al menos uno de los siguientes: una anotación de campo que contiene un límite 206 que indica que un campo 212 en el tipo de datos definido por el usuario 204 contiene un límite 218 para un procedimiento de acceso a la memoria 208; una anotación de procedimiento captador de límites 206 que indica que un procedimiento de obtención de límites 214 en el tipo de datos definido por el usuario 204 devuelve un límite 218 para el procedimiento de acceso a la memoria 208.

En algunas realizaciones, el sistema 102 incluye el código de lenguaje intermedio 124 que reside en la memoria, y el código 124 se anota con una representación de verificación de límites 220 de la anotación de verificación de límites definidos por el usuario 206.En algunas, el compilador 224 está configurado para insertar el código de verificación de límites 222 no solo para las anotaciones de verificación de límites definidos por el usuario 206, sino también para los tipos integrados 128. En algunas realizaciones, la anotación de verificación de límites definidos por el usuario 206 indica una verificación de límites complementaria, en el sentido de que el tipo definido por el usuario 204 envuelve un tipo integrado 128 que el compilador 224 está configurado para realizar una verificación de límites sin importar si hay presente cualquier anotación de verificación de límites definidos por el usuario 206.

En algunas realizaciones, los periféricos 106, tales como dispositivos de E/S para usuario humano (pantalla, teclado, ratón, tableta, micrófono, altavoz, sensor de movimiento, etc.) estarán presentes en comunicación operativa con uno o más procesadores 110 y memoria. Sin embargo, una realización también puede estar profundamente embebida en un sistema, de modo que ningún usuario humano 104 interactúe directamente con la realización. Los procesos de software pueden ser usuarios 104.

En algunas realizaciones, el sistema incluye múltiples ordenadores conectados por una red. El equipo de interfaz de red puede proporcionar acceso a las redes 108, utilizando componentes tales como una tarjeta de interfaz de red de conmutación de paquetes, un transceptor inalámbrico o una interfaz de red telefónica, por ejemplo, estará presente en un sistema de computación. Sin embargo, una realización también puede comunicarse a través del acceso directo a la memoria, medios no volátiles extraíbles u otros procedimientos de recuperación y/o transmisión de información, o una realización en un sistema de computación puede funcionar sin comunicarse con otros sistemas de computación.

Procesos

La Figura 3 ilustra algunas realizaciones de proceso en un diagrama de flujo 300. Los procesos mostrados en las Figuras se pueden realizar en algunas realizaciones automáticamente, por ejemplo, mediante un compilador 224 y un optimizador 226 bajo el control de una secuencia que requiere poca o ninguna entrada por parte del usuario, o por un generador de código fuente automático 122 que genera un tipo definido por el usuario 204 de acuerdo con las especificaciones suministradas por el usuario. Los procesos también se pueden realizar en parte de forma automática y en parte manualmente, a menos que se indique lo contrario. En una realización dada, se pueden repetir cero o más etapas ilustrados de un proceso, tal vez con diferentes parámetros o datos para operar. Las etapas en una realización también pueden realizarse en un orden diferente al orden de arriba a abajo que se presenta en la Figura 3. Las etapas se pueden realizar en serie, de manera parcialmente superpuesta, o totalmente en paralelo. El orden en el que se recorre el diagrama de flujo 300 es transversal para indicar que las etapas realizadas durante un proceso pueden variar de una ejecución del proceso a otra ejecución del proceso. El orden de recorrido del diagrama de flujo también puede variar de una realización del proceso a otra realización del proceso. Las etapas también se pueden omitir, combinar, renombrar, reagrupar o apartarse del flujo ilustrado, siempre que el proceso realizado sea operable y se ajuste a al menos una reivindicación.

Se proporcionan ejemplos en la presente memoria para ayudar a ilustrar aspectos de la tecnología, pero los ejemplos dados en la presente memoria no describen todas las posibles realizaciones. Las realizaciones no se limitan a las implementaciones, disposiciones, pantallas, características, enfoques o escenarios específicos proporcionados en la presente memoria. Una realización dada puede incluir características, mecanismos y/o estructuras de datos adicionales o diferentes, por ejemplo, y de lo contrario puede apartarse de los ejemplos proporcionados en la presente memoria.

Durante una etapa de identificación de tipo definido por el usuario 302, una realización identifica un tipo definido por el usuario 204 en un código fuente. La etapa 302 se puede llevar a cabo utilizando analizadores léxicos, analizadores y/u otros mecanismos, por ejemplo, adaptados para identificar los tipos definidos por el usuario 204 como se describe en la presente memoria. Específicamente, los mecanismos utilizados para reconocer las anotaciones familiares del código fuente pueden adaptarse para reconocer las anotaciones 206 por palabra clave.

Durante una etapa de inserción 304 de la representación de verificación de límites, una realización inserta una representación de verificación de límites 220 en el código de lenguaje intermedio 124 durante la compilación del código fuente anotado correspondiente. La etapa 304 se puede llevar a cabo utilizando árboles de análisis, árboles de sintaxis abstracta, atributos, vectores auxiliares generalizados y/u otros mecanismos, por ejemplo, adaptados para representar anotaciones de verificación de límites 206 como se describe en la presente memoria.

Durante una etapa de aplicación de optimización 306, una realización aplica optimización(es) 228 para reducir o eliminar la verificación de límites redundante. La optimización se puede aplicar al código fuente, al código intermedio y/o al código ejecutable, para reducir la verificación de límites duplicada que de lo contrario se produciría en el código ejecutable. La etapa 306 se puede llevar a cabo determinando analíticamente que un acceso a la memoria que está sujeto a la verificación de límites no puede asumir un valor durante la ejecución que daría lugar a un acceso a la memoria fuera de los límites permitidos. Por ejemplo, si un puntero se ha verificado con límites en un primer punto del código, y si los límites y el valor del puntero no pueden haber cambiado en un segundo punto más adelante en la ejecución del código, entonces no se necesita una verificación de límites en el segundo punto. Como otro ejemplo, si un puntero se ha verificado de límites en un primer punto del código, y si el valor del puntero no puede haber cambiado más en una dirección dada, los límites han cambiado en esa dirección en un segundo punto más adelante en la ejecución del código, entonces no se necesita verificación de límites en el segundo punto. Como otro ejemplo más, si un acceso a la memoria es inalcanzable por cualquier flujo de control durante la ejecución del código, entonces no se necesita una verificación de límites para ese acceso a la memoria.

En una etapa de inserción de código de verificación de límites 308, una realización inserta el código de verificación de límites 222 en el código ejecutable 126 durante la compilación del código fuente anotado correspondiente. Algunas realizaciones mantienen separados el código de lenguaje intermedio 124 y el código ejecutable 126, por ejemplo, en archivos separados, mientras que otras realizaciones combinan el código de lenguaje intermedio 124 y el código ejecutable 126. Por lo tanto, puede suceder que la etapa 308 inserta el código de verificación de límites 222 en el código ejecutable 126 que aparece en el mismo archivo o en el mismo bloque de memoria de trabajo

que el código de lenguaje intermedio 124. La etapa 308 se puede realizar utilizando árboles de análisis, árboles de sintaxis abstracta, selección de instrucciones, programación de instrucciones, asignación de registros y/u otros mecanismos, por ejemplo, adaptados para insertar el código de verificación de límites 222.

Durante una verificación de límites que complementa la etapa 310, una realización complementa la verificación de límites ya provista, como la verificación de límites de los tipos integrados, o la verificación de límites de tipos de matriz simples, por ejemplo. La etapa 310 se puede llevar a cabo definiendo un tipo 204 que tenga un tipo constituyente con verificación de límites, por ejemplo, o compilando dicho tipo. Por lo tanto, la etapa complementaria 310 puede ocurrir durante la inserción de representación de verificación de límites en la etapa 304 y/o durante la etapa de inserción de código de verificación de límites 308, si la inserción complementa la verificación de límites provista previamente. La etapa complementaria 310 también puede ser realizada por un desarrollador que define un tipo 204 que está anotado para agregar más límites a la verificación de límites previamente indicada.

Durante un tipo particular que define la etapa 312, un usuario define un tipo 204 que está libre de tipos de matriz 132, es decir, un tipo 204 que no tiene tipos de matriz como tipos constituyentes. Una clase 202 se considera un ejemplo de un tipo definido por el usuario 204. Una variable de un solo valor, como una variable de número entero, no se considera un caso especial de una matriz; para efectos de la etapa 312, las matrices tienen al menos dos elementos. La ausencia de matrices en los tipos definidos por la etapa 312 sirve para enfatizar la flexibilidad mejorada de la verificación de límites en tiempo de compilación como se describe en la presente memoria, en comparación con la familiar verificación de límites específicos de matriz. Los desarrolladores pueden utilizar herramientas de edición de código fuente y entornos de desarrollo 138 para recibir los tipos 204 definidos durante la etapa 312.

Durante una etapa de obtención de código fuente 314, un desarrollador o una realización que actúa en nombre de un desarrollador obtiene un código fuente 122.La etapa 314 se puede realizar utilizando sistemas de archivos, redes, entornos de desarrollo integrado 138 y/u otros mecanismos familiares.

Durante una etapa de especificación de tipo 316, un desarrollador o una realización que actúa en nombre de un desarrollador especifica un tipo definido por el usuario 204 (que puede ser, por ejemplo, una clase definida por el usuario 202) en el código fuente 122.Los desarrolladores pueden usar herramientas de edición de código fuente y entornos de desarrollo 138 para especificar los tipos 204 durante la etapa 316. En realizaciones particulares, la etapa 316 puede incluir definir la etapa 312 y/o la etapa complementaria 310.

Durante una etapa de localización de procedimiento 318, un desarrollador o una realización que actúa en nombre de un desarrollador ubica un procedimiento de acceso a la memoria 320 que está definido (por ejemplo, especificado 316) por un tipo definido por el usuario 204. Tales procedimientos 320 son ejemplos de código de acceso a la memoria en general. La etapa 318 se puede realizar utilizando herramientas de edición de código fuente y entornos de desarrollo 138, y en particular, las capacidades de búsqueda de palabras clave de estos.

Durante la(s) etapa(s) de anotación 322, un desarrollador o una realización que actúa en nombre de un desarrollador anota el código fuente para proporcionar al compilador 224 información de verificación de límites para la memoria que se asignará explícitamente para contener objetos u otras variables de un tipo definido por usuario 204.Por ejemplo, el código de acceso a la memoria puede anotarse con una anotación 206 de verificación de límites definidos por el usuario, lo que indica que el código accede a la memoria que está asignada explícitamente y/o sujeta a verificaciones de límites más allá de cualquier verificación es proporcionado por el entorno de lenguaje sin las anotaciones 206. El código que accede (o puede acceder) a la memoria que está asignada explícitamente puede identificarse al compilador 224 mediante una anotación de acceso asignada explícitamente 326 en la memoria 206. Las anotaciones 206 que proporcionan límites 210 se pueden colocar para anotar los mecanismos 322 que indican los límites, como por ejemplo al anotar 322 un campo 212 con una anotación 206 con límite de contenido campo 328, o al anotar 322 un procedimiento 214 con un procedimiento de captación de límites 330 de anotación 206.

En una etapa de envoltura 332, un desarrollador o una realización que actúa en nombre de un desarrollador envuelve un tipo existente en un tipo definido por el usuario 204.Es decir, el usuario define (especifica) un tipo 204 que tiene un tipo existente como un tipo constituyente, complementando de ese modo 310 la verificación de límite del tipo existente, si la hubiera. La etapa 332 se puede realizar utilizando herramientas de edición de código fuente y entornos de desarrollo 138.

Durante una segunda etapa que denota el límite 334, un desarrollador o una realización que actúa en nombre de un desarrollador denota una segunda condición del límite 218 para un tipo definido por el usuario 204.Es decir, el usuario complementa 310 la verificación de límites del tipo definido por el usuario especificando un límite diferente. Por ejemplo, un tipo puede tener un primer límite que refleja la memoria total asignada y también un segundo límite que refleja el uso real de la memoria asignada, por ejemplo, los registros que han sido marcados como "obsoletos" pueden considerarse fuera de límites incluso si residen en la memoria asignada para mantener registros. La etapa 334 puede realizarse utilizando herramientas y entornos de edición de código fuente 138.

En una etapa de reducción de verificación de límites duplicada, una realización reduce la verificación de límites duplicada, por ejemplo, aplicando 306 optimización(es) 228 que logran localizar y eliminar al menos una verificación de límites redundante.

Durante una etapa de compilación 338, una realización compila el código fuente anotado 322.La etapa 338 se puede llevar a cabo utilizando herramientas de compilación familiares y técnicas adaptadas para proporcionar la verificación de límites optimizados en tiempo de compilación de los tipos definidos por el usuario como se describe en la presente memoria.

Durante una etapa de configuración de memoria 340, un medio de memoria 112 está configurado por un tipo definido por el usuario 204, un compilador de optimización 224, 226 y/o de otro modo en relación con la verificación de límites optimizados en tiempo de compilación de tipos definidos por el usuario como discutido en la presente memoria.

Las etapas anteriores y sus interrelaciones se discuten con mayor detalle a continuación, en relación con diversas realizaciones.

Algunas realizaciones proporcionan un proceso para la verificación de límites en tiempo de compilación de tipos definidos por el usuario. El proceso incluye etapas realizadas durante la compilación de un programa desde un código fuente a través de un código de lenguaje intermedio hasta un código ejecutable. Se identifica una clase definida por el usuario 202 u otro tipo 204 en el código fuente 302. La clase puede estar destinada para acceder de forma segura a la memoria asignada explícitamente, por ejemplo. En algunas realizaciones, la clase definida por el usuario se define 312 como libre de cualquier tipo de matriz de elementos múltiples como un tipo constituyente. Una clase definida por el usuario puede tener como miembro anotado un código de acceso a la memoria 208 que está anotado 322 con una anotación 206 de verificación de límites definidos por el usuario 324. La clase 202 también puede tener como miembro anotado un miembro 210 que proporciona un límite que está anotado 322 para indicar que proporciona información de límite 218 para generar una verificación de límites en el código de acceso a la memoria. En respuesta al tipo anotado 204, se inserta una representación de verificación de límites 220 de la anotación de verificación de límites definidos por el usuario 304 en el código de lenguaje intermedio, y en algunos casos se aplica 306 una optimización 228 en un esfuerzo por reducir la verificación de límites duplicada que de lo contrario ocurriría en el código ejecutable.

En algunas realizaciones, el proceso incluye insertar 308 código de verificación de límites en el código de lenguaje intermedio en respuesta a representaciones de verificación de límites, y la etapa de aplicación aplica 306 la optimización al código de verificación de límites insertado en lugar de aplicar la optimización a la(s) representación(es) de verificación de límites. En algunas realizaciones, la etapa de aplicación aplica 306 la optimización a la(s) representación(es) de verificación de límites 220 en lugar de aplicar la optimización al código de verificación de límites 222.

En algunas realizaciones, la etapa de identificación identifica 302 un procedimiento 320 de acceso a la memoria que se ha anotado 322 con una anotación de acceso de memoria asignada explícitamente 326.En algunas, la etapa de identificación identifica 302 una anotación 206 que indica la verificación de límites definidos por el usuario, la cual pretende complementar 310 la verificación de límites definidos por el sistema de un tipo integrado 128. En algunas, la etapa de identificación identifica 302 una anotación 206 que indica la verificación de límites definidos por el usuario que pretende complementar 310 la verificación de límites definidos por el sistema de un tipo administrado por recolector de residuos 130.

Algunas realizaciones proporcionan un proceso para que un desarrollador de programas administre la verificación de límites en tiempo de compilación de tipos definidos por el usuario, es decir, los tipos que no están integrados. El proceso incluye obtener 314 un código fuente de un programa de ordenador, y especificar 316 un tipo de datos definido por el usuario en el código fuente (por ejemplo, escribiendo un tipo 204 o aceptando uno escrito previamente). El proceso también incluye ubicar 318 un procedimiento de acceso a la memoria 320 que está definido por el tipo de datos definido por el usuario, y anotar 322 el procedimiento de acceso a la memoria, por ejemplo, con accesos de memoria asignada explícitamente 326 u otra anotación 206. Además, el proceso incluye anotar 328 el código fuente con al menos uno de los siguientes: una anotación 206 de límite de contenido de campo 322 que indica que un campo 212 definido por el tipo de datos definido por el usuario 204 contiene un valor de límite 218 para procedimiento de acceso a la memoria, una anotación 206 del procedimiento de captación de límite 330 que indica que un procedimiento de captación de límite 214 definido por el tipo de datos definido por el usuario devuelve un valor de límite 218 para el procedimiento de acceso a la memoria.

En algunas realizaciones, el desarrollador anota 322 el código fuente con una anotación 206 que denota 334 un segundo límite para el procedimiento de acceso a la memoria. En algunas, el tipo definido por el usuario 204 envuelve 332 un tipo de matriz integrado 128, 132. En algunas, el tipo definido por el usuario 204 envuelve 332 un tipo administrado integrado 128, 130.

Medios configurados

Algunas realizaciones incluyen un medio de almacenamiento legible por ordenador configurado 112. El medio 112 puede incluir discos (magnéticos, ópticos o de otro tipo), RAM, EEPROMS u otras ROM, y/u otra memoria configurable, incluyendo en particular medios no transitorios legibles por ordenador (a diferencia de los cables y otros medios de señal propagada). El medio de almacenamiento que está configurado puede ser, en particular, un medio de almacenamiento extraíble 114 tal como un CD, DVD o memoria flash. Una memoria de propósito general, que puede ser removible o no, y puede ser volátil o no, puede configurarse en una realización utilizando elementos tales como los tipos definidos por el usuario 204 (incluidas sus anotaciones 206), y/u optimizadores 226 (que son adaptados para procesar anotaciones 206), en forma de datos 118 e instrucciones 116, leídos desde un medio extraíble 114 y/o desde otra fuente, como una conexión de red, para formar un medio configurado. El medio configurado 112 es capaz de hacer que un sistema de computación realice etapas del proceso para transformar el código fuente y otros a través de la anotación y la verificación de límites flexibles optimizados en tiempo de compilación como se describe en la presente memoria. Las Figuras 1 a 3 ayudan a ilustrar las realizaciones de medios de almacenamiento configurados y las realizaciones de procesos, así como las realizaciones de sistemas y procesos. En particular, cualquiera de las etapas del proceso ilustrados en la Figura 3, o enseñados de otro modo en la presente memoria, se pueden usar para ayudar a configurar un medio de almacenamiento para formar una realización del medio configurado.

Ejemplos adicionales

A continuación, se proporcionan detalles adicionales y consideraciones de diseño. Como con los otros ejemplos en la presente memoria, las características descritas se pueden usar individualmente y/o en combinación, o no en absoluto, en una realización dada.

Los expertos entenderán que los detalles de la implementación pueden pertenecer a un código específico, como las API específicas y los programas de muestra específicos, y por lo tanto no es necesario que aparezcan en cada realización. Los expertos también entenderán que los identificadores de programas y alguna otra terminología utilizada para analizar los detalles son específicos de la implementación y, por lo tanto, no tienen que ver con cada realización. No obstante, aunque no se requiere necesariamente que estén presentes aquí, estos detalles se proporcionan porque pueden ayudar a algunos lectores al proporcionar un contexto y/o pueden ilustrar algunas de las muchas implementaciones posibles de la tecnología que se analizan en la presente memoria.

Algunas realizaciones descritas en la presente memoria proporcionan los siguientes aspectos.

Primero, una forma de permitir que un programador defina un tipo de datos 204 (por ejemplo, una clase 202) para acceder a la memoria asignada explícitamente de una manera segura. El programador puede usar un conjunto de anotaciones 206, que el programador coloca en los procedimientos definidos por el tipo de datos. Un tipo de anotación 206 indica que el procedimiento anotado accede a la memoria asignada explícitamente y debe estar protegido por una verificación de límites. Otro tipo de anotación 206 indica que un campo 212 en el tipo de datos contiene el límite 218 en el acceso a la memoria. Un tercer tipo de anotación 206 indica que un procedimiento 214 en el tipo de datos devuelve el límite en el acceso a la memoria. Ya sea el segundo o el tercer tipo de anotación (o ambos) se pueden utilizar con una instancia particular del primer tipo de anotación.

En segundo lugar, un compilador 224 representa estas anotaciones 206 en su representación intermedia, es decir, en un código de lenguaje intermedio.

En tercer lugar, basándose en las anotaciones en la representación intermedia, el compilador 224 inserta las verificaciones de límite antes de las llamadas a los procedimientos 320 que acceden a la memoria asignada explícitamente.

En cuarto lugar, después de insertar verificaciones de límites 304/308, el compilador realiza una optimización que reduce 336 (y, por lo tanto, posiblemente elimina) verificaciones de límites innecesarios. Estas optimizaciones 228 pueden adaptarse para su uso en el presente contexto a partir de optimizaciones que son familiares en la literatura, extendiendo la optimización de matriz para comprender la representación intermedia anotada y eliminar las verificaciones de límites insertadas antes de las llamadas a procedimientos que acceden a la memoria asignada explícitamente, que es más complejo que simples matrices. El compilador 224 u otro optimizador 226 identifica comparaciones contra campos que contienen accesos de límite o procedimientos que devuelven accesos de límite, y las verificaciones insertadas antes de las llamadas a funciones y luego determina simbólicamente (mediante enfoques adaptados de técnicas familiares para eliminar las verificaciones de límites innecesarias en matrices) si las verificaciones pueden ser eliminadas de forma segura.

De esta manera, un programador puede usar la memoria asignada explícitamente de una manera relativamente segura. Esto permite un acceso eficiente y seguro a la memoria asignada explícitamente en el código

administrado.

En algunas realizaciones, el compilador 224 proporciona un conjunto de atributos que se pueden aplicar a cualquier estructura de datos, incluyendo estructuras de datos que no son meras matrices. De esta manera, tales realizaciones generalizan y agregan flexibilidad al trabajo anterior en la verificación de límites, por ejemplo, el trabajo que es específico para matrices como tipos de lenguaje integrados. Estas realizaciones permiten que el programador aplique la verificación de límites a estructuras de datos alternativas que están definidas por el programador y, en particular, permite la verificación de límites en situaciones en las que un compilador y un sistema de lenguaje no controlan el diseño de datos 216, o en las que el diseño de datos 216 puede ser arbitrario.

Algunas realizaciones se inspiran en la idea de "vectores auxiliares", que se utilizaron en la implementación de matrices en lenguajes de programación, y modifican el concepto para llegar a una realización en la que el programador puede definir la estructura de datos verificada de límites, en lugar de la estructura de datos definida por la implementación del lenguaje. Un vector auxiliar familiar contiene un puntero a un bloque de memoria que contiene los elementos de la matriz, los límites de la matriz y, posiblemente, otra información. Algunas realizaciones son o pueden integrarse con el trabajo familiar en la eliminación de la verificación de límites de la matriz, de modo que un programa se hace más eficiente tanto con respecto a la verificación de límites familiar de la matriz como con respecto a la verificación de límite flexible de tipo definido por el usuario 204 que se describe en la presente memoria.

En algunas realizaciones, las anotaciones 206 describen una verificación de corrección que se aplica en tiempo de ejecución si es necesario, no una propiedad semántica de alto nivel de una operación de biblioteca. El optimizador 226 intenta eliminar verificaciones innecesarias. En otro trabajo, por el contrario, un optimizador utiliza anotaciones para describir las propiedades semánticas de una biblioteca y optimizar el uso de las bibliotecas, no para reducir 336 controles de seguridad innecesarios como se describe en la presente memoria.

Algunas realizaciones tienen un entorno operativo 100 que contiene el Tiempo de Ejecución de Lenguaje Común (CLR), de Microsoft®, un tiempo de ejecución relativamente grande que incluye servicios y características tales como compilación en el momento justo (JIT), recolección de residuos (GC), reflexión en tiempo de ejecución y mucho más. Algunos tienen un entorno 100 que sigue muy de cerca el modelo C con la compilación tradicional (ahora a veces llamada con anticipación), aunque alguna GC puede proporcionarse para fines de seguridad de tipo.

En algunas realizaciones, C# permite anotar valores de retorno de forma que puedan usarse para permitir atributos en procedimientos:

```
[return: SomeAttribute]
int SomeMethod(){...}
```

En algunas realizaciones, las verificaciones de límites generadas por el compilador y eliminadas por el compilador están disponibles para, pero no se limitan a, estructuras de datos que tienen grupos indexados de recursos no administrados, típicamente memoria. Los programadores pueden anotar sus estructuras de datos para que un compilador 224 genere verificaciones de límites que se comportan en tiempo de ejecución de manera similar a las verificaciones de límites de la matriz (por ejemplo, al generar una excepción al violar un límite) y que pueden eliminarse mediante un enfoque familiar como el enfoque de Verificación de Límites de Matriz bajo Demanda (ABCD) o de optimizaciones familiares que reemplazan a la ABCD.

En algunas realizaciones, se proporcionan tres atributos personalizados.

Se aplica un atributo BoundsChecking a los procedimientos anotados 320.En respuesta, el compilador 224 insertará verificaciones de límites en los sitios de llamada a los procedimientos marcados como BoundsChecking. En una realización, el compilador 224 requiere que un procedimiento BoundsChecking tenga al menos un argumento, y requiere que el primer argumento sea del tipo Int32. La verificación de límites verificará que el primer argumento esté entre cero y el campo marcado como Bound (ver más abajo). En esta realización, todos los tipos con un procedimiento BoundsChecking tienen exactamente un campo Int32 marcado como Bound. La eliminación de las verificaciones de seguridad debidas a la adición de BoundsChecking debe considerarse un cambio importante.

Un atributo enlazado se aplica a los campos anotados 212.En una realización, el campo es un Int32 y será utilizado por la verificación de límites generada por el procedimiento BoundsChecking en el mismo tipo 204.

Se aplica un atributo BoundGetter a los procedimientos anotados 214.En una realización, si un procedimiento que devuelve un Bound no estará en línea, puede marcarse como BoundGetter, y las llamadas a él se tratarán como acceso al Bound.

En algunas realizaciones, el compilador 224 verificará los requisitos descritos anteriormente, pero es responsabilidad del programador asegurarse de que Bound solo se aplique a campos significativos y BoundGetter solo se aplique a los procedimientos que devuelven el Bound (o un valor menor que el Bound).En una realización que ha adaptado optimizaciones basadas en matrices para eliminar las verificaciones de límites, las verificaciones pueden eliminarse de manera poco segura si el campo Bound está mutado.

Algunas realizaciones, adoptan un enfoque basado en tipos, y esperan que los usuarios escriban un tipo que tenga forma de matriz, y le pidan al usuario que describa esa forma al compilador (donde es la longitud, donde está el descriptor de acceso). Pero los tipos estructurados no son requeridos en cada realización. En algunas realizaciones, las características involucradas incluyen el hecho de que una ubicación en el código de usuario requiere una verificación contra alguna variable de usuario y al compilador se le indica cómo construir esa verificación. Algunas realizaciones configuran las verificaciones disponibles para que parezcan verificaciones de matriz [0, longitud), como una opción de implementación.

Algunas realizaciones ponen [BoundsChecking] en los procedimientos 320.Algunos también los colocan directamente en el código fuente en otro código de acceso a la memoria 208, como en el siguiente ejemplo:

```
void Foo(int i) {
 byte* p = ...
  [BoundsChecking] (or [BoundsChecking(i)]
 ...*(p + i) ...
}
```

En la práctica, este tipo de anotación puede ser descartada por algunos estándares de lenguaje de código fuente. No se sigue en Lenguaje Intermedio de Microsoft (MSIL), que es una implementación de un estándar, ECMA 335, pero se puede seguir en otros lenguajes.

Algunas realizaciones no están restringidas a la protección de memoria asignada explícitamente. Por ejemplo, uno podría envolver 332 a una matriz administrada como tal:

```
class List {
 int[] arr = new int[20];
 [Bound]
 int count = 0;
 void Add(int i) {
  arr[count] = i;
  count = count + 1;
 }
 [BoundsChecking]
 void Get(int i) {
  return arr[i];
 }
}
```

En este ejemplo, el lenguaje proporciona verificaciones de límites de matriz existentes en arr, pero el desarrollador también desea verificaciones complementarias más sólidas, para garantizar no solo que i es menor que 20, sino también que es menor que la cantidad de elementos que se han añadido a la lista. Ambas verificaciones pueden ser candidatas para su eliminación a través de la verificación de límites de la matriz y/u otra optimización.

Exclusiones

Para ilustrar y aclarar adicionalmente las diferencias entre la verificación de límites en tiempo de compilación flexible descrita en la presente memoria y los enfoques anteriores, se proporciona la siguiente discusión de la verificación de límites de matriz familiar. Los conceptos y la tecnología descritos en esta discusión pueden ser compatibles en la práctica con las realizaciones que se enseñan en la presente memoria, ya que ninguno impide el uso del otro, sino que se encuentran fuera del alcance de las realizaciones para las cuales se busca protección en la presente memoria.

En un contexto de convergencia de atributos de límite, una aproximación anota atributos que pueden aparecer en un campo, parámetro o valor de retorno de puntero o matriz de C# o tipo de matriz:

[StaticBound(*n*)], donde *n* es un entero literal.
[BoundedBy(*ident*)], donde *ident* es un identificador que es:

Algún otro campo de tipo integral que sea miembro de la misma estructura que contiene inmediatamente,

o

Algún otro parámetro formal del mismo procedimiento/procedimiento, o

En el caso del valor de retorno, en realidad se adjunta al procedimiento.

Estos se pueden contraer en un solo atributo si se permite que el argumento se defina como {string|int}; puede que no necesitemos un segundo nombre de atributo.

Si el campo/parámetro que lleva estos atributos es un puntero, la presencia del atributo conlleva un contrato en el que se *deben* verificar las operaciones de indexación a través del puntero. Desde una perspectiva social, se puede observar que es más fácil retroceder de manera compatible que ir hacia otro lado.

Un atributo que puede aparecer en un parámetro, campo o valor de retorno que actúa como un índice (que debe ser de algún tipo integral):
[Range(*begin,end*)], donde *begin* y *end* pueden ser enteros literales o identificadores codificados como cadenas, y la expectativa normal es que el comienzo será la constante literal cero.

Cuando este atributo aparece en un parámetro formal, indica que el llamador debe verificar/descargar por rango el parámetro actual.

Cuando este atributo aparece en un campo, indica que el RHS de asignación o inicialización debe ser verificado o descargado por rango de manera equivalente.

Cuando este atributo aparece en un procedimiento, indica un requisito sobre el valor de retorno que el procedimiento
debe verificar/descargar antes de la devolución.

Con respecto a las burbujas de versión, a través de la mutilación y la envoltura es posible ser compatible hacia abajo con llamadores inconscientes si eso resulta deseable.

El rango se puede capturar con mayor precisión como:
[Range(inclusiveBase,ExclusiveBound)]

En cuanto a las ventajas y desventajas del límite exclusivo, una desventaja puede ser la incapacidad de codificar (por ejemplo) MAXINT para enteros. La alternativa [*Range(inclusiveBase,InclusiveBound*)] se convertiría casi invariablemente en un caso de uso de la forma: [Range(0,*boundIdent*-1)], que parece incómodo y plantea el problema de las expresiones en los atributos.

Ese problema se puede resolver de manera directa mediante el uso de un atributo diferente en los casos en que el valor máximo representable debe incluirse en el rango:
[*AtLeast*(*lowerBound*)]or[GreaterThanOrEqualTo(*lowerBound*)]

Dejando el tipo de parámetro subyacente para especificar el límite superior implícitamente en virtud del hecho de que cada tipo de C# tiene inherentemente un límite de rango como consecuencia de su tipo.

Los atributos BoundedBy y Range se pueden desacoplar.

Se puede especificar un atributo para tomar "int o string" en una posición de parámetro dada. Uno puede escribir varios constructores para un atributo personalizado, por lo que puede tomar diferentes tipos en la misma posición, o hacer esto con parámetros nombrados.

Con esto concluye la discusión de las exclusiones.

Conclusión

Aunque las realizaciones particulares se ilustran y describen expresamente en la presente memoria como procesos, como medios configurados o como sistemas, se apreciará que la discusión de un tipo de realización también generalmente se extiende a otros tipos de realización. Por ejemplo, las descripciones de los procesos en conexión con la Figura 3 también ayudan a describir los medios configurados y ayudan a describir el funcionamiento de los sistemas y fabricantes como los que se analizan en relación con otras Figuras. No se sigue que las limitaciones de una realización sean necesariamente leídas en otra. En particular, los procesos no están necesariamente limitados a las estructuras de datos y las matrices presentados mientras se discuten los sistemas o fabricantes como las memorias configuradas.

No todos los elementos mostrados en las figuras necesitan estar presentes en cada realización. A la inversa, una realización puede contener artículos que no se muestran expresamente en las Figuras. Aunque algunas posibilidades se ilustran aquí en el texto y las figuras mediante ejemplos específicos, las realizaciones pueden apartarse de estos ejemplos. Por ejemplo, las características específicas de un ejemplo se pueden omitir, cambiar de nombre, agrupar de manera diferente, repetir, crear instancias en hardware y/o software de manera diferente, o ser una combinación de características que aparecen en dos o más de los ejemplos. La funcionalidad mostrada en una ubicación también puede proporcionarse en una ubicación diferente en algunas realizaciones.

Se ha hecho referencia a las figuras mediante números de referencia. Cualquier inconsistencia aparente en la redacción asociada con un número de referencia dado, en las figuras o en el texto, debe entenderse como simplemente ampliando el alcance de lo que hace referencia ese número.

Como se usa en la presente memoria, términos tales como "un", "uno", "una", "el" y "la" incluyen uno o más del elemento o paso indicado. En particular, en las reivindicaciones, una referencia a un artículo generalmente significa que al menos uno de dichos artículos está presente y una referencia a una etapa significa que se realiza al menos una instancia de la etapa.

Los subtítulos se proporcionan únicamente por conveniencia; la información sobre un tema dado se puede encontrar fuera de la sección cuyo subtítulo indica ese tema.

Todas las reivindicaciones tal como se presentan son parte de la especificación.

Aunque se han mostrado realizaciones ejemplares en las figuras y se han descrito anteriormente, será evidente para los expertos en la técnica que pueden realizarse numerosas modificaciones sin apartarse de los principios y conceptos expuestos en las reivindicaciones, y que tales modificaciones no necesitan abarcar un concepto abstracto completo. Si bien el tema se describe en un lenguaje específico para características estructurales y/o actos de procedimiento, debe entenderse que el tema definido en las reivindicaciones adjuntas no está necesariamente limitado a las características o actos específicos descritos anteriormente. No es necesario que cada medio o aspecto identificado en una definición o ejemplo dado esté presente o sea utilizado en cada realización. Por el contrario, las características y los actos específicos descritos se describen como ejemplos a tener en cuenta al implementar las reivindicaciones.

Todos los cambios que no lleguen a abarcar una idea abstracta completa, pero que estén dentro del significado y el rango de equivalencia de las reivindicaciones, deben incluirse dentro de su alcance en la medida en que lo permita la ley.

REIVINDICACIONES

1. Un medio de almacenamiento no transitorio legible por ordenador configurado con datos y con instrucciones que cuando son ejecutados en al menos un procesador hacen que el(los) procesador(es) realice(n) un proceso para verificar límites en tiempo de compilación de tipos definidos por el usuario (204), comprendiendo el proceso las siguientes etapas realizadas durante la compilación de un programa desde un código fuente a través de un código de lenguaje intermedio hasta un código ejecutable:

identificar en el código fuente (122) una clase definida por el usuario (202) para acceder de forma segura a la memoria, es decir, una clase definida por el usuario (202) que tiene como miembro anotado un código de acceso a la memoria que está anotado con una anotación de verificación de límites definidos por el usuario (206) y también tiene como otro miembro anotado un miembro que proporciona un límite que se anota para indicar que proporciona información de límite para generar una verificación de límites en el código de acceso a la memoria;

la anotación de verificación de límites definidos por el usuario (206) comprende una anotación que indica al compilador que el código accede a un búfer mapeado en memoria u otra memoria asignada explícitamente;

insertar (304) en el código de lenguaje intermedio (124) una representación de verificación de límites (220) de la anotación de verificación de límites definidos por el usuario (206), siendo realizada la inserción durante la compilación del código fuente anotado correspondiente y siendo lograda la inserción (304) utilizando al menos uno de árboles de análisis, árboles de sintaxis abstracta, atributos y vectores auxiliares generalizados para representar la anotación de verificación de límites definidos por el usuario;

aplicar, al código de lenguaje intermedio y/o al código ejecutable, una optimización (228) que reduce la verificación de límites duplicada que de otro modo ocurriría en el código ejecutable (126);

en el que la aplicación de la optimización comprende determinar que todos los accesos dentro de un bucle a una variable de la clase definida por el usuario (204) están dentro de los límites de dirección de memoria permitidos de la variable y, en respuesta a esto, eliminar múltiples verificaciones de límites que de otro modo ocurrirían como un resultado de la ejecución del bucle, en el que la optimización se logra determinando analíticamente que un acceso a la memoria, que está sujeto a la verificación de límites, no puede asumir un valor durante la ejecución, lo que daría lugar a un acceso a la memoria fuera de los límites permitidos.

2. El medio configurado según la reivindicación 1, que además comprende insertar el código de verificación de límites (222) en el código de lenguaje intermedio (124) en respuesta a las representaciones de verificación de límites (220), y en el que, en la etapa de aplicación, la optimización se aplica al código de verificación de límites insertado (222) en lugar de aplicar la optimización a la(s) representación(es) de verificación de límites (220).

3. El medio configurado según la reivindicación 1, en el que, en la etapa de aplicación, la optimización (228) se aplica a la(s) representación(es) de verificación de límites (220) en lugar de aplicar la optimización (228) al código de verificación de límites (222) en el código de lenguaje intermedio (124).

4. El medio configurado según la reivindicación 1, en el que en la etapa de identificación identifica un procedimiento de acceso a la memoria anotado con una anotación de accesos de memoria asignada explícitamente.

5. El medio configurado según la reivindicación 1, en el que en la etapa de identificación identifica una anotación que indica la verificación de límites definidos por el usuario para complementar la verificación de límites definidos por el sistema de un tipo integrado.

6. El medio configurado según la reivindicación 1, en el que en la etapa de identificación identifica una anotación que indica la verificación de límites definidos por el usuario para complementar la verificación de límites definidos por el sistema de un tipo administrado por recolector de residuos.

7. Un sistema de computación que comprende:

un procesador lógico (110);

una memoria en comunicación operable con el procesador lógico;

un código fuente (122) que reside en la memoria y tiene un tipo definido por el usuario (204), el tipo definido por el usuario (202) tiene un procedimiento de acceso a la memoria que se anota con una anotación de verificación de límites definidos por el usuario (206), teniendo también el tipo definido por el usuario (204) al menos un especificador de límite;

la anotación de verificación de límites definidos por el usuario (206) comprende una anotación que indica al compilador que el código accede a un búfer mapeado en memoria u otra memoria asignada explícitamente;

un compilador (224) que reside en la memoria y está configurado para insertar en un código de lenguaje intermedio (124) una representación de verificación de límites (220) de la anotación de verificación de límites definidos por el usuario (206), la inserción se realiza durante la compilación del código fuente anotado correspondiente, y la inserción (304) se realiza utilizando al menos uno de árboles de análisis, árboles de sintaxis abstracta, atributos y vectores auxiliares generalizados para representar la anotación de verificación de límites definidos por el usuario;

un optimizador (226) que reside en la memoria y está configurado para aplicar, al código de lenguaje intermedio y/o al código ejecutable, una optimización (228) para reducir la verificación de límites duplicada;

en el que la aplicación de la optimización comprende determinar que todos los accesos dentro de un bucle a una variable de la clase definida por el usuario (204) están dentro de los límites de dirección de memoria permitidos de la variable y, en respuesta a esto, eliminar múltiples verificaciones de límites que de otro modo ocurrirían como un resultado de la ejecución del bucle, en el que la optimización se logra determinando analíticamente que un acceso a la memoria, que está sujeto a la verificación de límites, no puede asumir un valor durante la ejecución, lo que daría lugar a un acceso a la memoria fuera de los límites permitidos.

8. El sistema según la reivindicación 7, en el que el código fuente anotado (122) comprende un código fuente de controlador de dispositivo, y el tipo definido por el usuario corresponde a un búfer mapeado en memoria.

9. El sistema según la reivindicación 7, en el que el código fuente comprende tipos de datos recolectados de residuos, y el tipo definido por el usuario corresponde a la memoria asignada explícitamente.

10. El sistema según la reivindicación 7, en el que el tipo definido por el usuario se define como libre de cualquier tipo de matriz de elementos múltiples como un tipo constituyente.

11. El sistema según la reivindicación 7, en el que el(los) especificador(es) vinculado(s) comprende(n) al menos uno de los siguientes:

una anotación de campo que contiene un límite que indica que un campo en el tipo de datos definido por el usuario contiene un límite para el procedimiento de acceso a la memoria;

una anotación de procedimiento de captador de límites que indica que un procedimiento de obtención de límites en el tipo de datos definido por el usuario devuelve un límite para el procedimiento de acceso a la memoria.

12. El sistema según la reivindicación 7, que además comprende un código de lenguaje intermedio que reside en la memoria y que se anota con una representación de verificación de límites (220) de la anotación de verificación de límites definidos por el usuario (206).

13. El sistema según la reivindicación 7, en el que el tipo definido por el usuario tiene un diseño de datos que no está controlado por el compilador (224).

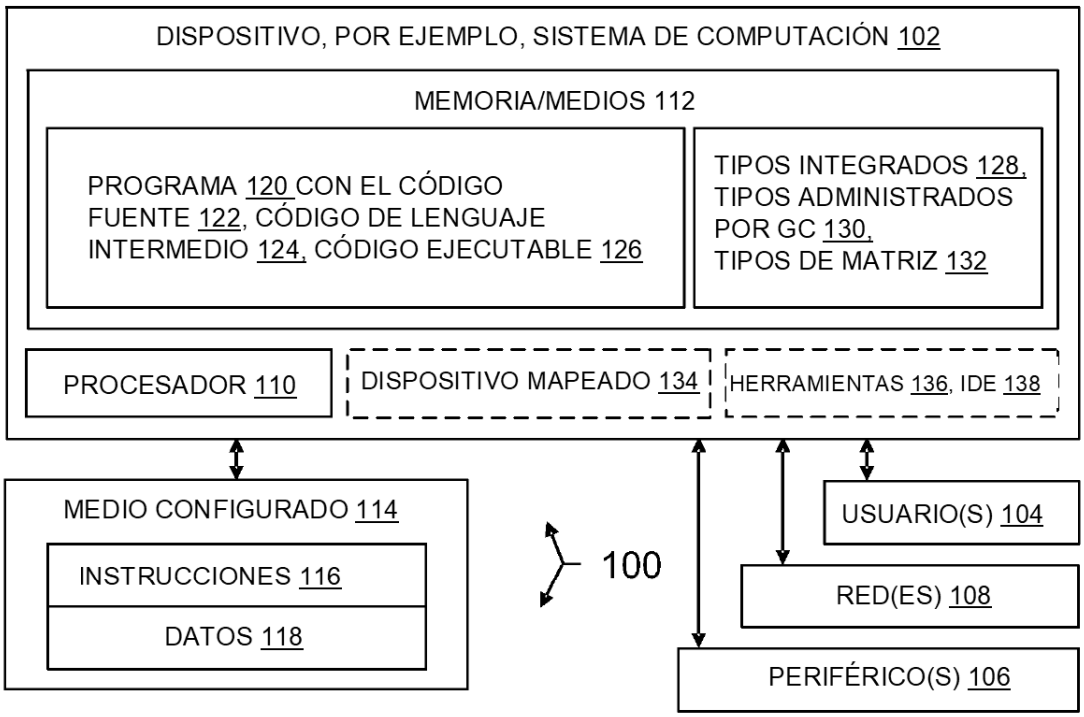
DISPOSITIVO, POR EJEMPLO, SISTEMA DE COMPUTACIÓN 102
MEMORIA/MEDIOS 112
PROGRAMA 120 CON EL CÓDIGO FUENTE 122, CÓDIGO DE LENGUAJE INTERMEDIO 124, CÓDIGO EJECUTABLE 126
TIPOS INTEGRADOS 128, TIPOS ADMINISTRADOS POR GC 130, TIPOS DE MATRIZ 132
PROCESADOR 110
DISPOSITIVO MAPEADO 134
HERRAMIENTAS 136, IDE 138
MEDIO CONFIGURADO 114
INSTRUCCIONES 116
DATOS 118
100
USUARIO(S) 104
RED(ES) 108
PERIFÉRICO(S) 106

Fig. 1

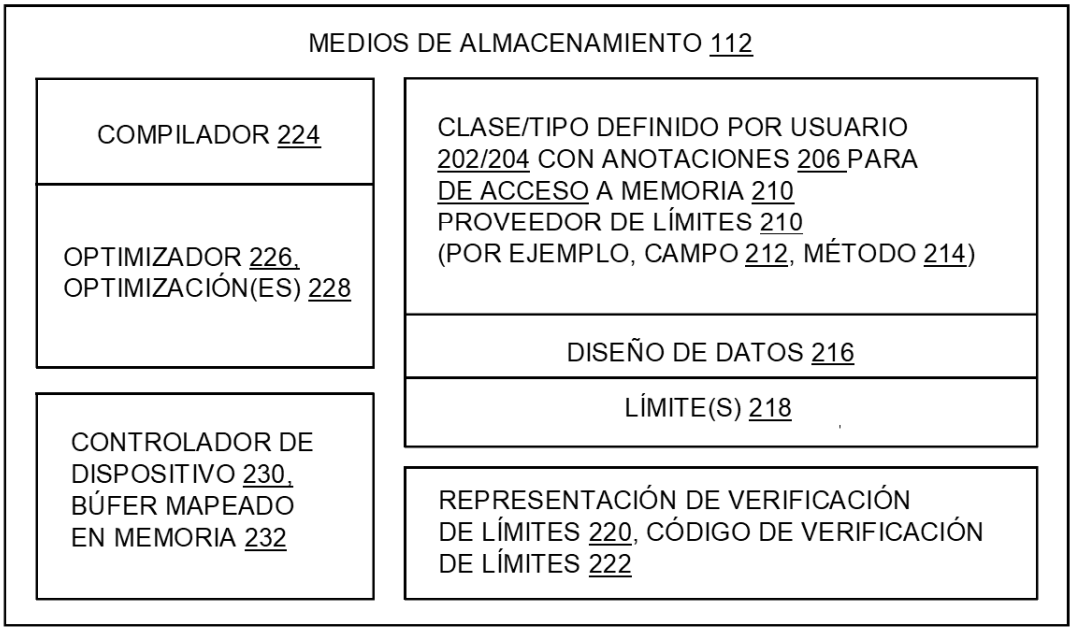
MEDIOS DE ALMACENAMIENTO 112
COMPILADOR 224
OPTIMIZADOR 226, OPTIMIZACIÓN(ES) 228
CONTROLADOR DE DISPOSITIVO 230, BÚFER MAPEADO EN MEMORIA 232
CLASE/TIPO DEFINIDO POR USUARIO 202/204 CON ANOTACIONES 206 PARA DE ACCESO A MEMORIA 210
PROVEEDOR DE LÍMITES 210 (POR EJEMPLO, CAMPO 212, MÉTODO 214)
DISEÑO DE DATOS 216
LÍMITE(S) 218
REPRESENTACIÓN DE VERIFICACIÓN DE LÍMITES 220, CÓDIGO DE VERIFICACIÓN DE LÍMITES 222

Fig. 2


300
IDENTIFICAR 302 CLASE/TIPO DEFINIDO POR EL USUARIO
INSERTAR 304 REPRESENTACIÓN DE VERIFICACIÓN DE LÍMITES
APLICAR 306 OPTIMIZACIÓN(ES)
INSERTAR 308 CÓDIGO DE VERIFICACIÓN DE LÍMITES
COMPLEMENTAR 310 LA VERIFICACIÓN EXISTENTE DE LÍMITES
DEFINIR 312 COMO LIBRE DE TIPO(S) DE MATRIZ
OBTENER 314 CÓDIGO FUENTE
ESPECIFICAR 314 CLASE/TIPO DEFINIDO POR EL USUARIO
LOCALIZAR 318 MÉTODO DE ACCESO A LA MEMORIA 320
ANOTAR 322 EL CÓDIGO FUENTE, POR EJEMPLO, CON ANOTACIÓN(ES) PARA VERIFICACIÓN DE LÍMITES DEFINIDOS POR EL USUARIO 324, ACCESOS DE MEMORIA ASIGNADA EXPLÍCITAMENTE 326, LÍMITES DE CONTENIDO DE CAMPO 328, MÉTODO DE OBTENCIÓN DE LÍMITES 330
ENVOLVER 332 UN TIPO EXISTENTE
DENOTAR 334 SEGUNDO LÍMITE
REDUCIR 336 VERIFICACIÓN DE LÍMITES DUPLICADA
COMPILAR 338 EL CÓDIGO FUENTE ANOTADO
CONFIGURAR 340 MEMORIA

Fig. 3