(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
14 February 2008 (14.02.2008)

PCT

(10) International Publication Number
WO 2008/018962 A1

(51) International Patent Classification:
G06F 9/45 (2006.01) *G06F 9/00* (2006.01)

(21) International Application Number:
PCT/US2007/015404

(22) International Filing Date: 3 July 2007 (03.07.2007)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
11/499,191 4 August 2006 (04.08.2006) US

(71) Applicant: **MICROSOFT CORPORATION** [US/US]; One Microsoft Way, Redmond, WA 98052-6399 (US).

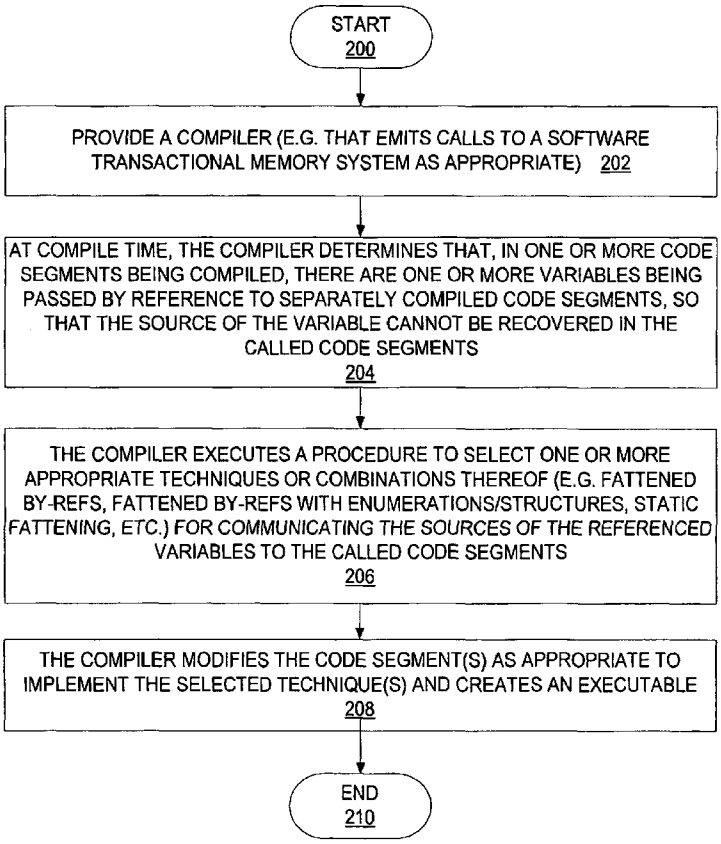
(72) Inventors: **DUFFY, John, Joseph**; c/o Microsoft Corporation, One Microsoft Way, Redmond, WA 98052-6399 (US). **MAGRUDER, Michael, M.**; c/o Microsoft Corporation, One Microsoft Way, Redmond, WA 98052-6399 (US). **GRAEFE, Goetz**; c/o Microsoft Corporation, One Microsoft Way, Redmond, WA 98052-6399 (US). **DETLEFS, David**; c/o Microsoft Corporation, One Microsoft Way, Redmond, WA 98052-6399 (US).

(81) Designated States *(unless otherwise indicated, for every kind of national protection available)*: AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BH, BR, BW, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DO, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, GT, HN, HR, HU, ID, IL, IN, IS, JP, KE, KG, KM, KN, KP, KR, KZ, LA, LC, LK, LR, LS, LT, LU, LY, MA, MD, ME, MG, MK, MN, MW, MX, MY, MZ, NA, NG, NI, NO, NZ, OM, PG, PH, PL, PT, RO, RS, RU, SC, SD, SE, SG, SK, SL, SM, SV, SY, TJ, TM, TN, TR, TT, TZ, UA, UG, UZ, VC, VN, ZA, ZM, ZW.

(84) Designated States *(unless otherwise indicated, for every kind of regional protection available)*: ARIPO (BW, GH, GM, KE, LS, MW, MZ, NA, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European (AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI,

[Continued on next page]

(54) Title: SOFTWARE TRANSACTIONAL PROTECTION OF MANAGED POINTERS



(57) Abstract: Various technologies and techniques are disclosed that provide software transactional protection of managed pointers. A software transactional memory system interacts with and/or includes a compiler. At compile time, the compiler determines that there are one or more reference arguments in one or more code segments being compiled whose source cannot be recovered. The compiler executes a procedure to select one or more appropriate techniques or combinations thereof for communicating the sources of the referenced variables to the called code segments to ensure the referenced variables can be recovered when needed. Some examples of these techniques include a fattened by-ref technique, a static fattening technique, a dynamic ByRefInfo type technique, and others. One or more combinations of these techniques can be used as appropriate.

WO 2008/018962 A1

FR, GB, GR, HU, IE, IS, IT, LT, LU, LV, MC, MT, NL, PL, PT, RO, SE, SI, SK, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

Declarations under Rule 4.17:

— *as to applicant's entitlement to apply for and be granted a patent (Rule 4.17(ii))*

— *as to the applicant's entitlement to claim the priority of the earlier application (Rule 4.17(iii))*

Published:

— *with international search report*

— *before the expiration of the time limit for amending the claims and to be republished in the event of receipt of amendments*

SOFTWARE TRANSACTIONAL PROTECTION OF MANAGED POINTERS

BACKGROUND

[001] Software transactional memory (STM) is a concurrency control mechanism analogous to database transactions for controlling access to shared memory in concurrent computing. A transaction in the context of transactional memory is a piece of code that executes a series of reads and writes to shared memory. A data value in the context of transactional memory is the particular segment of shared memory being accessed, such as a single object, a cache line (such as in C++), a page, a single word, etc.

[002] A software transactional memory system must implement transactional semantics for all memory operations, otherwise atomicity and isolation are compromised. Object-based STMs use a per-object concurrency control mechanism: each object contains "metadata" used by the STM for concurrency control (a kind of lock). This requires that, at the point of invoking a transactional operation on a field of an object, the object's identity can be discovered. But even object-based systems have data not stored in objects: e.g., static and local variables. STMs may transct such data in a different fashion. Unfortunately, in a software system that permits taking an address of a data item and passing this address to separately compiled functional units, often the source of the data item cannot be recovered. As an example, consider this C# code:

```
class Clss {
   int m_fld;
   static int s_fld;
}

void a() {
   int j = 7;
   Data d = new Clss();
   int[] arr = new int[1];
   b(ref j);
   b(ref d.m_fld);
   b(ref Clss.s_fld);
```

```
    b(ref arr[0]);
  }

  void b(ref int x) {
    atomic {
      x++;
    }
  }
```

[003] This code examples above illustrate the classic problem. When method b is compiled, the runtime argument that will be supplied for the parameter x is not known. Yet the caller, a, calls b with four different types of values, each of which refers to a type of location which utilizes different concurrency control mechanisms. Generally, object-based STM systems ensure transactional semantics for static variables, local variables (locals and arguments), and instance fields or array elements in different ways, meaning b must somehow recover the source of the argument x.

SUMMARY

[004] Various technologies and techniques are disclosed that provide software transactional protection of managed pointers. A software transactional memory system interacts with and/or includes a compiler. At compile time, the compiler determines that there are one or more reference arguments in one or more code segments being compiled whose source cannot be recovered. The compiler executes a procedure to select one or more appropriate techniques or combinations thereof for communicating the sources of the referenced variables to the called code segments to ensure the referenced variables can be recovered when needed. Some examples of these techniques include a fattened by-ref technique, a static fattening technique, a dynamic ByRefInfo type technique, and others. One or more combinations of these techniques can be used as appropriate.

[005] This Summary was provided to introduce a selection of concepts in a simplified form that are further described below in the Detailed Description. This Summary is not intended to identify key features or essential features of the claimed subject matter, nor is it intended to be used as an aid in determining the scope of the claimed subject matter.

BRIEF DESCRIPTION OF THE DRAWINGS

[006] Figure 1 is a diagrammatic view of a computer system of one implementation.

[007] Figure 2 is a high-level process flow diagram for one implementation of the system of Figure 1.

[008] Figure 3 is a process flow diagram for one implementation of the system of Figure 1 illustrating the stages involved in using a fattened by-ref technique to allow the identity of the object(s) to be recovered.

[009] Figure 4 is a diagram that includes a code segment illustrating a code segment typed by a developer in a programming language.

[010] Figure 5 is a diagram that includes a code segment of one implementation that illustrates how the code segment of Figure 4 is modified in the called function using the fattened by-ref technique described in Figure 3.

[011] Figure 6 is a diagram that includes a code segment of one implementation that illustrates how the calling function is modified using the fattened by-ref technique described in Figure 3.

[012] Figure 7 is a process flow diagram for one implementation of the system of Figure 1 illustrating the stages involved in using a fattened by-ref with enumerations/structures technique to allow the identity of the object(s) to be recovered.

[013] Figure 8 is a diagram that includes a code segment illustrating a definition of a by-ref argument descriptor for one implementation as used in the technique described in Figure 7.

[014] Figure 9 is a process flow diagram for one implementation of the system of Figure 1 illustrating the stages involved in using a static fattening technique to allow the identity of the object(s) to be recovered.

[015] Figure 10 is a diagram of one implementation that illustrates some exemplary values that can be included in an enumeration for a StaticByRefInfo structure.

[016] Figure 11 is a diagram that of one implementation that illustrates exemplary stack frame contents when the techniques in Figure 9 are used.

[017] Figure 12 is a process flow diagram for one implementation of the system of Figure 1 that illustrates the stages involved in using a dynamic ByRefInfo type technique to allow the identity of the object(s) to be recovered for unusual control flow situations.

[018] Figure 13 is a diagram that includes a code segment illustrating a code segment typed by a developer in a programming language.

[019] Figure 14 is a diagram that includes a code segment of one implementation that illustrates how the code segment of Figure 13 is modified using the dynamic ByRefInfo type technique described in Figure 12.

[020] Figure 15 is a process flow diagram for one implementation of the system of Figure 1 that illustrates the stages involved in using a combination of techniques to allow the identity of the two objects to be recovered when needed.

DETAILED DESCRIPTION

[021] For the purposes of promoting an understanding of the principles of the invention, reference will now be made to the embodiments illustrated in the drawings and specific language will be used to describe the same. It will nevertheless be understood that no limitation of the scope is thereby intended. Any alterations and further modifications in the described embodiments, and any further applications of the principles as described herein are contemplated as would normally occur to one skilled in the art.

[022] The system may be described in the general context as a software transactional memory system, but the system also serves other purposes in addition to these. In one implementation, one or more of the techniques described herein can be implemented as features within a framework program such as MICROSOFT® .NET Framework, or from any other type of program or service that provides platforms for developers to develop software applications. In another implementation, one or more of the techniques described herein are implemented as features with other applications that deal with developing applications that execute in concurrent environments.

[023] As shown in Figure 1, an exemplary computer system to use for implementing one or more parts of the system includes a computing device, such as computing device 100. In its most basic configuration, computing device 100 typically includes at least one processing unit 102 and memory 104. Depending on the exact configuration and type of computing device, memory 104 may be volatile (such as RAM), non-volatile (such as ROM, flash memory, etc.) or some combination of the two. This most basic configuration is illustrated in Figure 1 by dashed line 106.

[024] Additionally, device 100 may also have additional features/functionality. For example, device 100 may also include additional storage (removable and/or non-removable) including, but not limited to, magnetic or optical disks or tape. Such

additional storage is illustrated in Figure 1 by removable storage 108 and non-removable storage 110. Computer storage media includes volatile and nonvolatile, removable and non-removable media implemented in any method or technology for storage of information such as computer readable instructions, data structures, program modules or other data. Memory 104, removable storage 108 and non-removable storage 110 are all examples of computer storage media. Computer storage media includes, but is not limited to, RAM, ROM, EEPROM, flash memory or other memory technology, CD-ROM, digital versatile disks (DVD) or other optical storage, magnetic cassettes, magnetic tape, magnetic disk storage or other magnetic storage devices, or any other medium which can be used to store the desired information and which can accessed by device 100. Any such computer storage media may be part of device 100.

[025] Computing device 100 includes one or more communication connections 114 that allow computing device 100 to communicate with other computers/applications 115. Device 100 may also have input device(s) 112 such as keyboard, mouse, pen, voice input device, touch input device, etc. Output device(s) 111 such as a display, speakers, printer, etc. may also be included. These devices are well known in the art and need not be discussed at length here. In one implementation, computing device 100 includes software transactional memory application 200 and compiler application 202. In one implementation, compiler application 202 uses the software transactional memory application 200 to generate properly transacted code.

[026] Turning now to Figures 2-14 with continued reference to Figure 1, the stages for implementing one or more implementations of software transactional memory application 200 are described in further detail. In one form, the process of Figure 2 is at least partially implemented in the operating logic of computing device 100. The procedure begins at start point 200 with providing a compiler (e.g. that emits calls to a

software transactional memory system as appropriate) (stage 202). At compile time, the compiler determines that, in one or more code segments being compiled, there are one or more variables being passed by reference to separately compiled code segments, which prevents the source of those references from being identified or recovered in the called code segments (stage 204). The compiler executes a procedure to select one or more appropriate techniques or combinations thereof (e.g. fattened by-refs, fattened by-refs with enumerations/structures, static fattening, etc.) for communicating the sources of these references from the calling code segments to the called code segments (stage 206). The compiler modifies the code segment(s) as appropriate to implement the selected technique(s) and creates an executable (stage 208). The process ends at end point 210.

[027] Figure 3 illustrates one implementation of the stages involved in using a fattened by-ref technique to allow the identity of the object(s) to be recovered. In one form, the process of Figure 3 is at least partially implemented in the operating logic of computing device 100. The procedure begins at start point 230 with determining that a fattened by-ref technique should be used to allow the sources of the reference arguments to be recovered when needed (stage 232). A new type (e.g. struct ByRefArgDesc{...}) is created that contains sufficient information to distinguish the different kinds of actual arguments and to provide the transactional memory system sufficient information to employ appropriate methods of implementing transactional semantics for accesses to the reference argument in the called code function or method (stage 234). For each function or method that has at least one by-ref argument, change each by-ref argument [e.g. foo(...ref tt t,...)] to a pair that includes the new type [e.g. foo(...ref ByRefArgDesc brifor_t, ref tt t,...)] (stage 236). At the call site for each function or method that was changed to the pair, insert code to assign the appropriate value(s) to the new type, and add

the type as a parameter to the call to the function or method (stage 238). The process ends at end point 240.

[028] Turning now to Figure 4-6, some exemplary code segments are illustrated to show how the code typed by the developer is modified based upon the fattened by-ref techniques discussed in Figure 3. Figure 4 is a diagram that includes a code segment illustrating a code segment 260 typed by a developer in a programming language. Figure 5 is a diagram that includes a code segment 262 of one implementation that illustrates how the code segment of Figure 4 is modified in the called function using the fattened by-ref technique described in Figure 3. Note how the extra argument (ref ByRefArgDesc briFor_t) is inserted in the call to foo. Turning now to Figure 6, the calling function 264 for foo is shown as modified using the fattened by-ref technique described in Figure 3. Note how a value is assigned to the ByRefArgDesc bri_t variable 266, and the value is then passed as an argument to foo 268.

[029] Figure 7 illustrates one implementation of the stages involved in using a variant of the fattened by-ref technique described above, with enumerations/structures technique to allow the sources of the reference arguments to be recovered. In this variant, when a function or method has several by-ref arguments, their respective ByRefArgDesc descriptors are gathered together into a single composite data structure (a ByRefInfoHolder), and this data structure is passed by reference to the called function or method, adding only a single extra argument instead of one extra argument for each reference argument.

[030] In one form, the process of Figure 7 is at least partially implemented in the operating logic of computing device 100. The procedure begins at start point 290 with determining that a fattened by-ref with enumerations/structures technique should be used to allow the source (e.g. identity) of reference arguments to be recovered when needed

(stage 292). At each call site that invokes the particular function or method that has by-ref arguments whose sources require identification, allocate a ByRefInfoHolder of a size sufficient to hold the information for all the by-ref arguments of the invoked function (stage 294). The ByRefInfoHolder is an instance of one of several pre-defined value types, one for several different numbers of by-ref arguments, up to some maximum (stage 296). If the number of by-ref arguments in a method exceeds this maximum, several such ByRefInfoHolders may be allocated, and passed as multiple arguments, or each ByRefInfoHolder can refer to the next in turn. The ByRefInfoHolder is passed by reference, and used when transacting accesses to the argument in the callee, or it is ignored in non-transactional code (stage 298). If a first function calls a second function with a by-ref argument, and the second function passes that argument along in a call to a third function, then the second function will stack-allocate a ByRefInfoHolder for the number of reference arguments in the second call, and copy into it, at that the appropriate argument position, the information that the first function passed it (stage 300). The process ends at end point 302.

[031] Figure 8 is a diagram that includes a code segment illustrating a definition of a by-ref argument descriptor for one implementation as used in the techniques described in Figure 7. The code segment includes an enumeration called ByRefArgKind 310, a structure called ByRefArgDesc 312, and a structure called ByRefInfoHolder2 314. In the ByRefArgDesc, the contents of the corresponding m_data field depend on the kind. For BRAK_Obj, the by-ref argument is an interior pointer into some heap object, and the field contains the offset of the by-ref in the object. This case covers object fields and array elements (of any dimension). For the BRAK_TMW, the field contains a Transactional Memory Word (TMW), which is used for concurrency control. In one implementation, the TMW contains a version, a count of pessimistic readers, and an exclusive writer

indicator. This covers (at least) statics. For the BRAK_NoTMW, the contents of the field are irrelevant, since the by-ref does not require transactional primitives. This covers locals, and may also cover by-refs in calls from unmanaged code back to managed.

[032] Figure 9 illustrates one implementation of the stages involved in using a static fattening technique to allow the identity of the object(s) to be recovered. In one form, the process of Figure 9 is at least partially implemented in the operating logic of computing device 100. The procedure begins at start point 350 with determining that a static fattening technique should be used to allow the source of reference arguments to be recovered when needed (stage 352). The runtime maintains a global table associating addresses of procedure calls and reference argument positions with information about the source of that argument in the calling method; such information is called a StaticByRefInfo (stage 354). Entries in this table are created at compile time; when compiling a call to a method with by-ref arguments, for each such by-ref argument, the compiler creates a StaticByRefInfo describing the source of the actual argument at this argument position, and stores this StaticByRefInfo into the global table (indexed by the address of the call sites and the argument position) (stage 356). When a method is being executed, and an access involving a by-ref argument is encountered, the system knows which of the method's arguments are being accessed, and the address of the instruction that invoked the currently-executing method (via that method's return address) (stage 360). The system can use these two pieces of information to index the global table to discover the StaticByRefInfo that describes the source of the reference argument (stage 362). The process ends at end point 364.

[033] Figure 10 is a diagram of one implementation that illustrates some exemplary values that can be included in an enumeration for a StaticByRefInfo structure. In one implementation, a StaticByRefInfo structure contains two arguments: the first one

being just an identifier that is used to indicates the type of argument (e.g. local, heap, static, etc.), and the second one being the auxiliary data that describes the argument in further detail. For example, if the argument was a local variable, then the structure 380 might contain <1,0>. In that example, the 1 indicates that it is a local variable (e.g. 1 = local variable), and the 0 is just auxiliary data that is not really needed in this particular scenario. If the argument was a field of a heap object, then the structure 382 might contain the number two for the first position (e.g. 2 = heap object), and the second position might contain the offset value of the field within the object. If the argument was a static, then the structure might contain a number three for the first position (e.g. 3 = static), and the second position might contain the TMW address. If the argument was an array element, then the structure 386 might contain a number 4 for the first position (e.g. 4 = array element), and the second position might contain a stack frame offset value, indicating a location in the caller's stack frame that will contain the array reference. If the argument was itself a by-ref argument, then the structure 390 might contain a five (e.g. 5 = by-ref argument) for the first position, and the second position might contain the by-ref argument position.

[034] Figure 11 is a diagram 400 of one implementation that illustrates exemplary stack frame contents when the techniques in Figure 9 are used. A caller method 402 calls a callee 404. The caller method has a local variable a containing a reference to an array of integers. The variable a is held in a stack slot (406) located 48 bytes above the stack pointer in the caller stack frame. The caller passes a reference to the 4^{th} element of that array to the callee. In compiling the caller, the compiler puts an entry <4, 48> in the global table to describe this reference argument. This indicates that the reference is to an element of an array, and that a reference to the head of the array (required by the

transactional memory implementation), may be found by looking 48 bytes above the bottom of the caller's stack frame.

[035] Figure 12 illustrates one implementation of the stages involved in using a dynamic ByRefArgDesc type technique to allow the identity of the object(s) to be recovered for unusual control flow situations. In one form, the process of Figure 12 is at least partially implemented in the operating logic of computing device 100. The procedure begins at start point 410 with determining that a dynamic ByRefArgDesc type technique should be used to allow the sources of reference variables to be recovered when needed (such as when value of an object is not known until runtime since it is contained in a code segment inside an IF statement that changes its reference type) (stage 412). The system modifies the code segment to create an instance of ByRefArgDesc type to describe each of the values assigned to the object (e.g. in the IF statement), and "flow" them with the object (stage 414). The system records in the global table that the by-ref info for the method's argument at this call site is held in a dynamic ByRefArgDesc structure in the stack frame of the caller of the method, and records in the global table the offset of that dynamic ByRefArgDesc structure (stage 416). By creating dynamic by-ref information in this way, unusual control flow problems can be solved (stage 418). The process ends at end point 420.

[036] Turning now to Figures 13-14, code examples are shown that illustrate how code typed by a developer is modified based upon the techniques described in Figure 12 to allow the object(s) to be recovered in unusual control flow situations. Figure 13 is a diagram that includes a code segment 440 typed by a developer in a programming language. The code segment 440 includes an IF statement that changes the value of t1 to two different types of values depending on the outcome (442 and 444, respectively). Figure 14 shows how the code segment 440 is modified according to the techniques of

Figure 12. For example, a dynamic ByRefArgDesc object is declared in the beginning of the procedure (called dbri4t1). The kind and val properties are then set for the object depending on the direction taken in the IF statements (448, 450, 452, and 454, respectively).

[037] Figure 15 is a process flow diagram for one implementation of the system of Figure 1 that illustrates the stages involved in using a combination of techniques to allow the identity of the two objects to be recovered when needed. In one form, the process of Figure 15 is at least partially implemented in the operating logic of computing device 100. The procedure begins at start point 470 determining at compile time that there is a first argument and a second argument referenced in at least one code segment being compiled whose source (identity) cannot be recovered (stage 472). The system selects at least one technique for communicating with a source of the first argument (stage 474), and selects at least one technique for communicating with a source of the second argument (stage 476). The system modifies the at least one code segment to implement the first technique (e.g. a static fattening technique, etc.) for the first argument, and the second technique (e.g. a dynamic by-ref info type technique, etc.) for the second argument (stage 478). The second argument has a problem because its value is not known until runtime (stage 478). An executable is created using the modified code segment (stage 480). The process ends at end point 482.

[038] Although the subject matter has been described in language specific to structural features and/or methodological acts, it is to be understood that the subject matter defined in the appended claims is not necessarily limited to the specific features or acts described above. Rather, the specific features and acts described above are disclosed as example forms of implementing the claims. All equivalents, changes, and modifications

that come within the spirit of the implementations as described herein and/or by the following claims are desired to be protected.

[039] For example, a person of ordinary skill in the computer software art will recognize that the client and/or server arrangements, and/or data layouts as described in the examples discussed herein could be organized differently on one or more computers to include fewer or additional options or features than as portrayed in the examples.

What is claimed is:

1. A method for providing software transactional protection of managed pointers comprising the steps of:

at compile time, determining that there are one or more variables being passed by reference to separately compiled code segments in at least one code segment being compiled (204);

selecting at least one technique for communicating the sources of the variables being passed by reference (206);

modifying the at least one code segment to implement the selected technique (208); and

creating an executable using the modified code segment (208).

2. The method of claim 1, wherein the at least one technique is a fattened by-ref technique (232), and wherein the technique comprises:

creating a new by-ref info type that contains information necessary to distinguish different kinds of actual arguments (234);

for a method that has at least one by-ref argument, change the by-ref argument to include an additional argument for a variable using the new by-ref info type (236); and

at a call site for the method that has the at least one by-ref argument, inserting code to assign an appropriate one or more values to the new by-ref info type, and adding the type as a call site argument to the call to the method (238).

3. The method of claim 1, wherein the at least one technique is a modified fattened by-ref technique that uses enumerations and structures (292), and wherein the technique comprises:

at each call site that invokes a particular method that has reference arguments needing identified, allocating a by-ref information holder (294); and

passing the by-ref information holder by reference (298).

4. The method of claim 3, wherein the by-ref information holder is an instance of one of a plurality of pre-defined value types (296).

5. The method of claim 1, wherein the at least one technique is a static fattening technique (352), and wherein the technique comprises:

creating a global table for storing static by-ref information (354); and

when compiling a call to a method accepting one or more by-ref arguments, adding static by-ref information entries to the global table describing the actual arguments to the one or more by-ref arguments, for later retrieval (356).

6. The method of claim 5, further comprising:

at runtime, when a method is being executed and an access involving a by-ref argument is encountered, indexing a global table to discover the static by-ref information (360).

7. The method of claim 6, wherein the indexing of the global table is achieved by using knowledge of an address of an instruction that invoked the method (360).

8. The method of claim 1, wherein the at least one technique comprises a plurality of techniques that are used in combination (296).

9. The method of claim 1, wherein technique is selected from the group consisting of a fattened by-ref technique, a modified fattened by-ref technique, and a static fattening technique (206).

10. A computer-readable medium having computer-executable instructions for causing a computer to perform the steps recited in claim 1.

11. A method for providing software transactional protection of managed pointers comprising the steps of:

determining that a static fattening technique should be used to allow a source of a particular reference argument to be recovered when needed (352);

at compile time, creating a static data structure for storing static by-ref information about the particular reference argument (356);

creating a global table for storing static by-ref information (354); and

when compiling a call to a method accepting the particular reference argument, adding static by-ref information entries to the global table describing the actual arguments to the particular reference argument, for later retrieval (356).

12. The method of claim 11, further comprising:

at runtime, when a method is being executed, and an access involving the particular reference argument is encountered, determining an address of the call site that invoked the method by a return address of the method (360).

13. The method of claim 12, wherein the return address is used to index a global table in which static by-ref info is stored (360).

14. The method of claim 13, further comprising:

locating a calling stack frame in which a caller recorded an actual reference or other information describing a by-ref argument at a fixed known offset in that stack frame (386).

15. A computer-readable medium having computer-executable instructions for causing a computer to perform the steps recited in claim 11.

16. A computer-readable medium having computer-executable instructions for causing a computer to perform steps comprising:

at compile time, determine that there is a first argument and a second argument referenced in at least one code segment being compiled whose source cannot be recovered (472);

select at least one technique for communicating a first source of the first argument (474);

select at least one technique for communicating a second source of the second argument, the second technique solving a problem that is only present with the second argument and not the first argument (476); and

modify the at least one code segment to implement the first technique for the first argument and the second technique for the second argument (478).

17. The computer-readable medium of claim 16, wherein the first technique is implemented using a static fattening technique (478).

18. The computer-readable medium of claim 16, wherein the problem that is only present with the second argument is that a value of the second argument is not known until runtime (478).

19. The computer-readable medium of claim 18, wherein the problem is solved using a dynamic by-ref info type technique (478).

20. The computer-readable medium of claim 16, further having computer-executable instructions for causing a computer to perform the step comprising:

create an executable using the modified code segment (480).

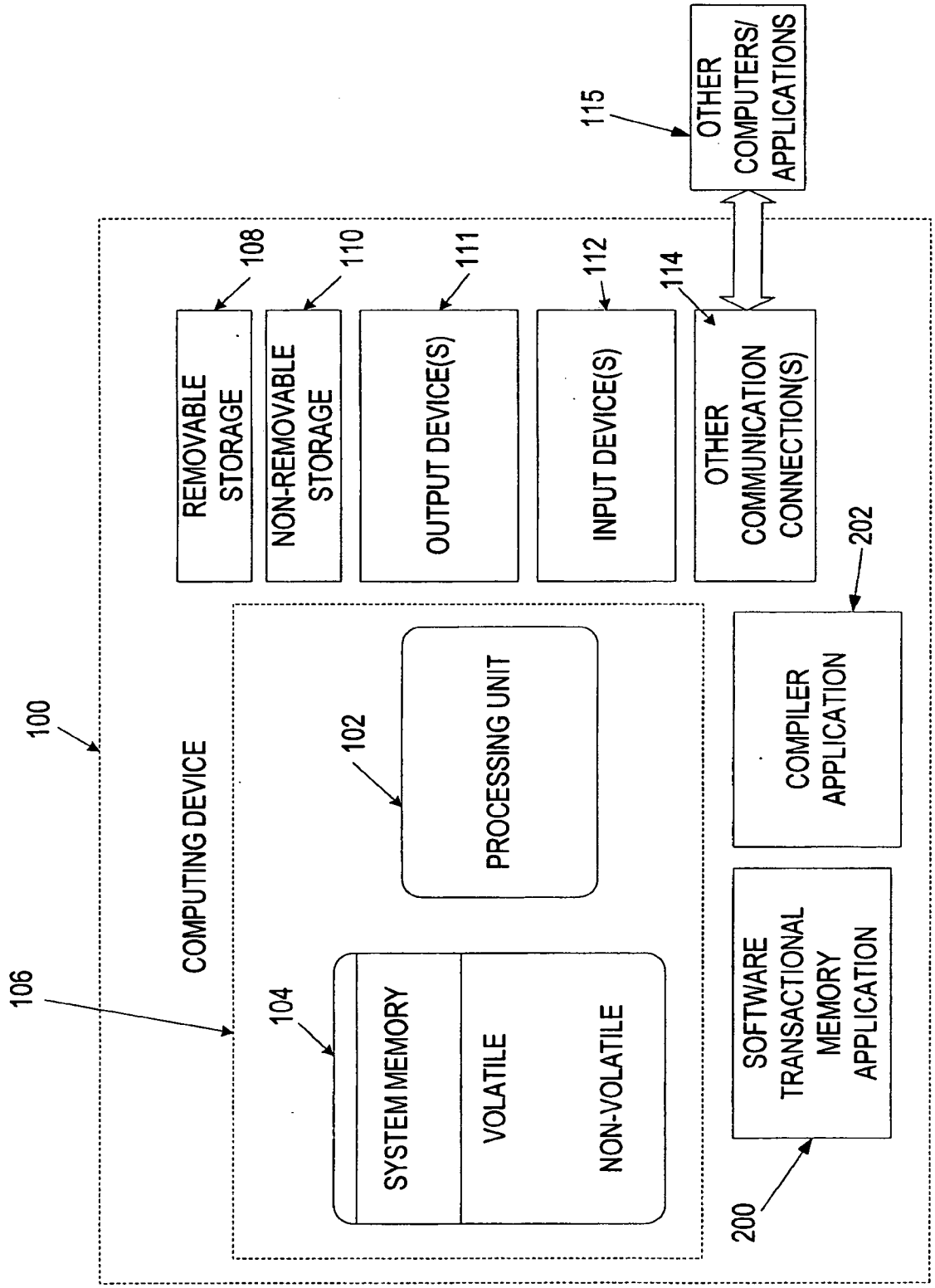

100
COMPUTING DEVICE
106
102
PROCESSING UNIT
104
SYSTEM MEMORY
VOLATILE
NON-VOLATILE
108
REMOVABLE STORAGE
110
NON-REMOVABLE STORAGE
111
OUTPUT DEVICE(S)
112
INPUT DEVICE(S)
114
OTHER COMMUNICATION CONNECTION(S)
115
OTHER COMPUTERS/ APPLICATIONS
202
COMPILER APPLICATION
200
SOFTWARE TRANSACTIONAL MEMORY APPLICATION

FIG. 1

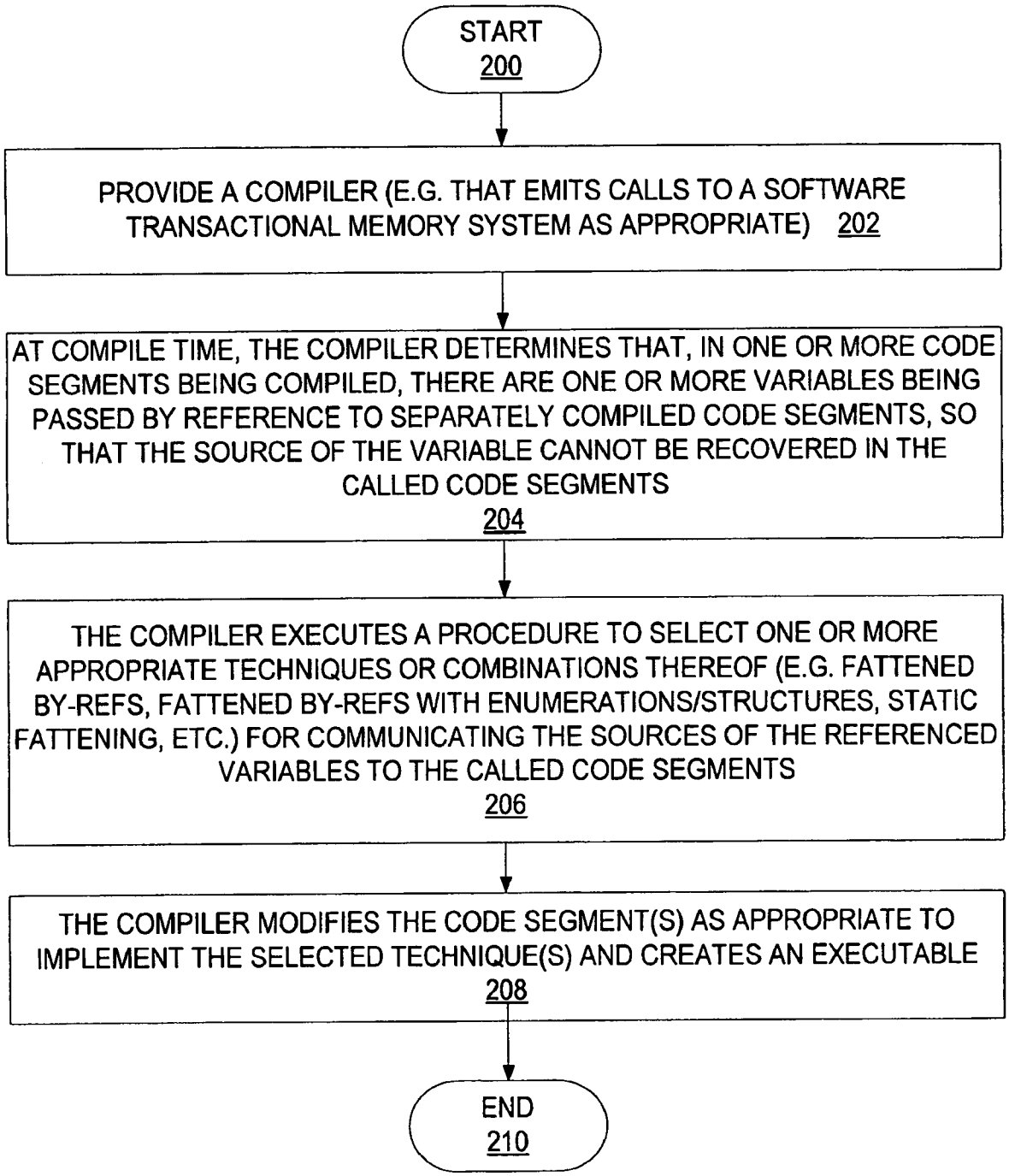

START
200
PROVIDE A COMPILER (E.G. THAT EMITS CALLS TO A SOFTWARE TRANSACTIONAL MEMORY SYSTEM AS APPROPRIATE) 202
AT COMPILE TIME, THE COMPILER DETERMINES THAT, IN ONE OR MORE CODE SEGMENTS BEING COMPILED, THERE ARE ONE OR MORE VARIABLES BEING PASSED BY REFERENCE TO SEPARATELY COMPILED CODE SEGMENTS, SO THAT THE SOURCE OF THE VARIABLE CANNOT BE RECOVERED IN THE CALLED CODE SEGMENTS
204
THE COMPILER EXECUTES A PROCEDURE TO SELECT ONE OR MORE APPROPRIATE TECHNIQUES OR COMBINATIONS THEREOF (E.G. FATTENED BY-REFS, FATTENED BY-REFS WITH ENUMERATIONS/STRUCTURES, STATIC FATTENING, ETC.) FOR COMMUNICATING THE SOURCES OF THE REFERENCED VARIABLES TO THE CALLED CODE SEGMENTS
206
THE COMPILER MODIFIES THE CODE SEGMENT(S) AS APPROPRIATE TO IMPLEMENT THE SELECTED TECHNIQUE(S) AND CREATES AN EXECUTABLE
208
END
210

FIG. 2

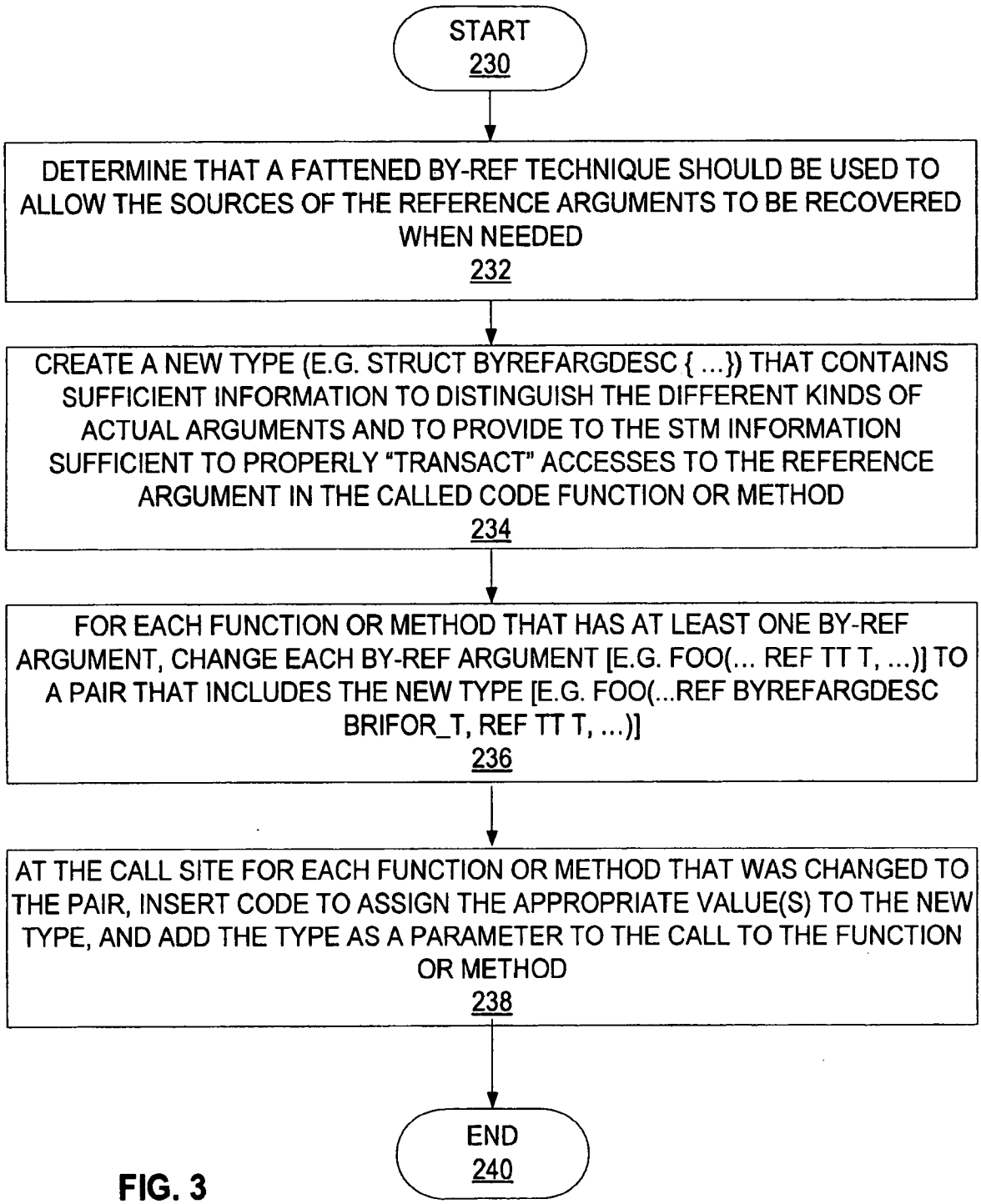

START
230
DETERMINE THAT A FATTENED BY-REF TECHNIQUE SHOULD BE USED TO ALLOW THE SOURCES OF THE REFERENCE ARGUMENTS TO BE RECOVERED WHEN NEEDED
232
CREATE A NEW TYPE (E.G. STRUCT BYREFARGDESC { ...}) THAT CONTAINS SUFFICIENT INFORMATION TO DISTINGUISH THE DIFFERENT KINDS OF ACTUAL ARGUMENTS AND TO PROVIDE TO THE STM INFORMATION SUFFICIENT TO PROPERLY "TRANSACT" ACCESSES TO THE REFERENCE ARGUMENT IN THE CALLED CODE FUNCTION OR METHOD
234
FOR EACH FUNCTION OR METHOD THAT HAS AT LEAST ONE BY-REF ARGUMENT, CHANGE EACH BY-REF ARGUMENT [E.G. FOO(... REF TT T, ...)] TO A PAIR THAT INCLUDES THE NEW TYPE [E.G. FOO(...REF BYREFARGDESC BRIFOR_T, REF TT T, ...)]
236
AT THE CALL SITE FOR EACH FUNCTION OR METHOD THAT WAS CHANGED TO THE PAIR, INSERT CODE TO ASSIGN THE APPROPRIATE VALUE(S) TO THE NEW TYPE, AND ADD THE TYPE AS A PARAMETER TO THE CALL TO THE FUNCTION OR METHOD
238
END
240

FIG. 3

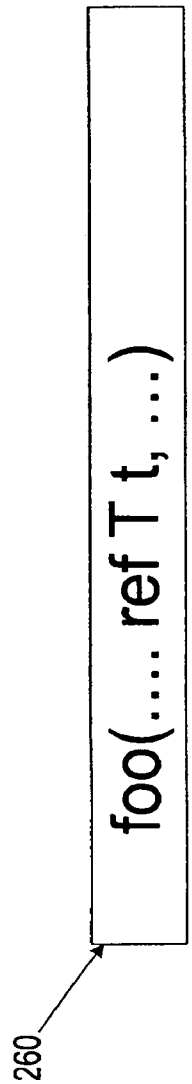


FIG. 4

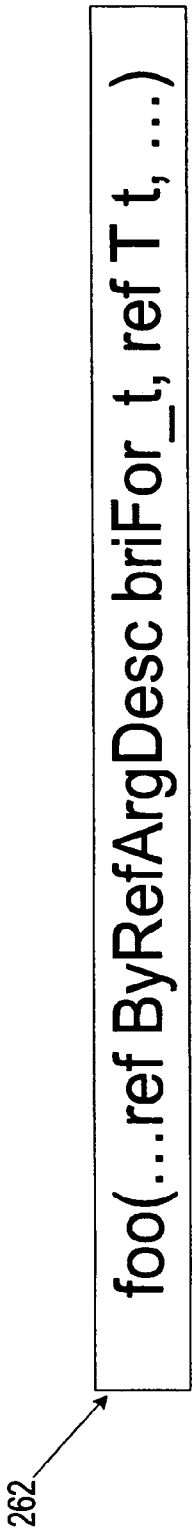


FIG. 5

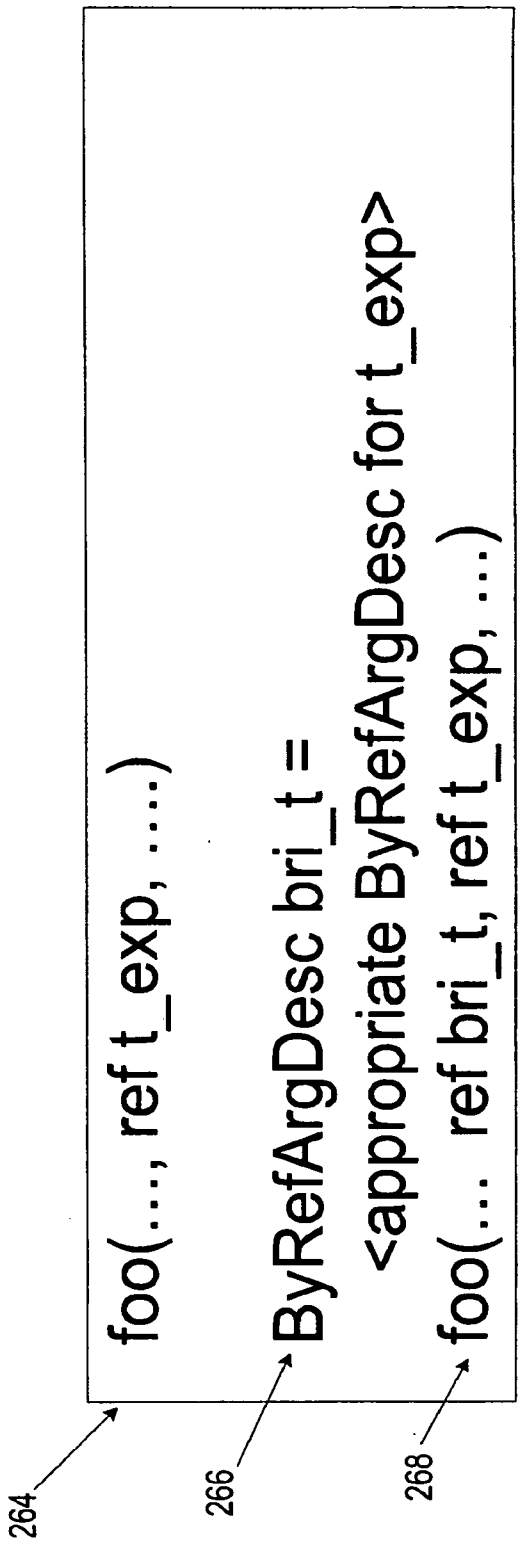


FIG. 6

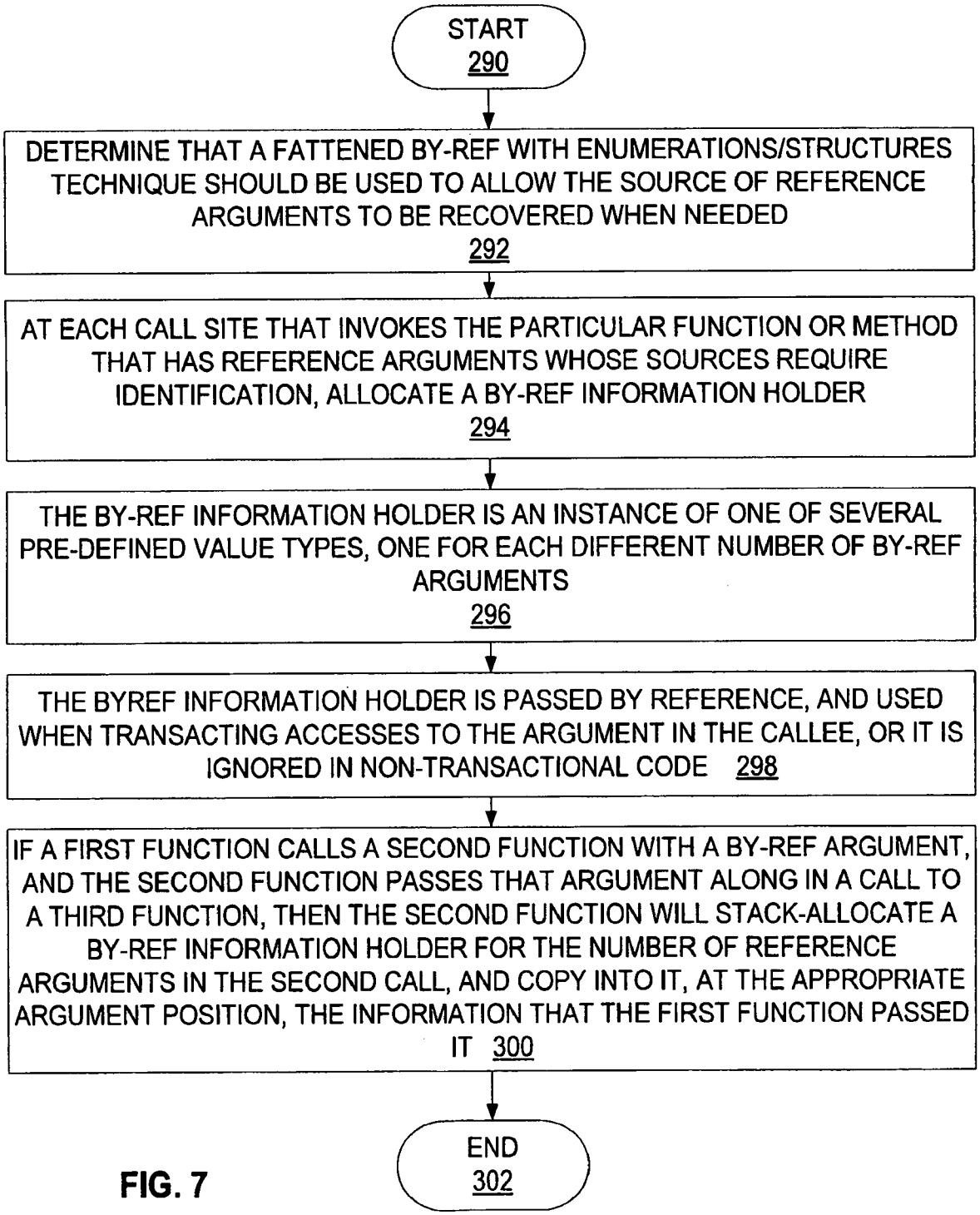

START
290
DETERMINE THAT A FATTENED BY-REF WITH ENUMERATIONS/STRUCTURES TECHNIQUE SHOULD BE USED TO ALLOW THE SOURCE OF REFERENCE ARGUMENTS TO BE RECOVERED WHEN NEEDED
292
AT EACH CALL SITE THAT INVOKES THE PARTICULAR FUNCTION OR METHOD THAT HAS REFERENCE ARGUMENTS WHOSE SOURCES REQUIRE IDENTIFICATION, ALLOCATE A BY-REF INFORMATION HOLDER
294
THE BY-REF INFORMATION HOLDER IS AN INSTANCE OF ONE OF SEVERAL PRE-DEFINED VALUE TYPES, ONE FOR EACH DIFFERENT NUMBER OF BY-REF ARGUMENTS
296
THE BYREF INFORMATION HOLDER IS PASSED BY REFERENCE, AND USED WHEN TRANSACTING ACCESSES TO THE ARGUMENT IN THE CALLEE, OR IT IS IGNORED IN NON-TRANSACTIONAL CODE 298
IF A FIRST FUNCTION CALLS A SECOND FUNCTION WITH A BY-REF ARGUMENT, AND THE SECOND FUNCTION PASSES THAT ARGUMENT ALONG IN A CALL TO A THIRD FUNCTION, THEN THE SECOND FUNCTION WILL STACK-ALLOCATE A BY-REF INFORMATION HOLDER FOR THE NUMBER OF REFERENCE ARGUMENTS IN THE SECOND CALL, AND COPY INTO IT, AT THE APPROPRIATE ARGUMENT POSITION, THE INFORMATION THAT THE FIRST FUNCTION PASSED IT 300
END
302

FIG. 7

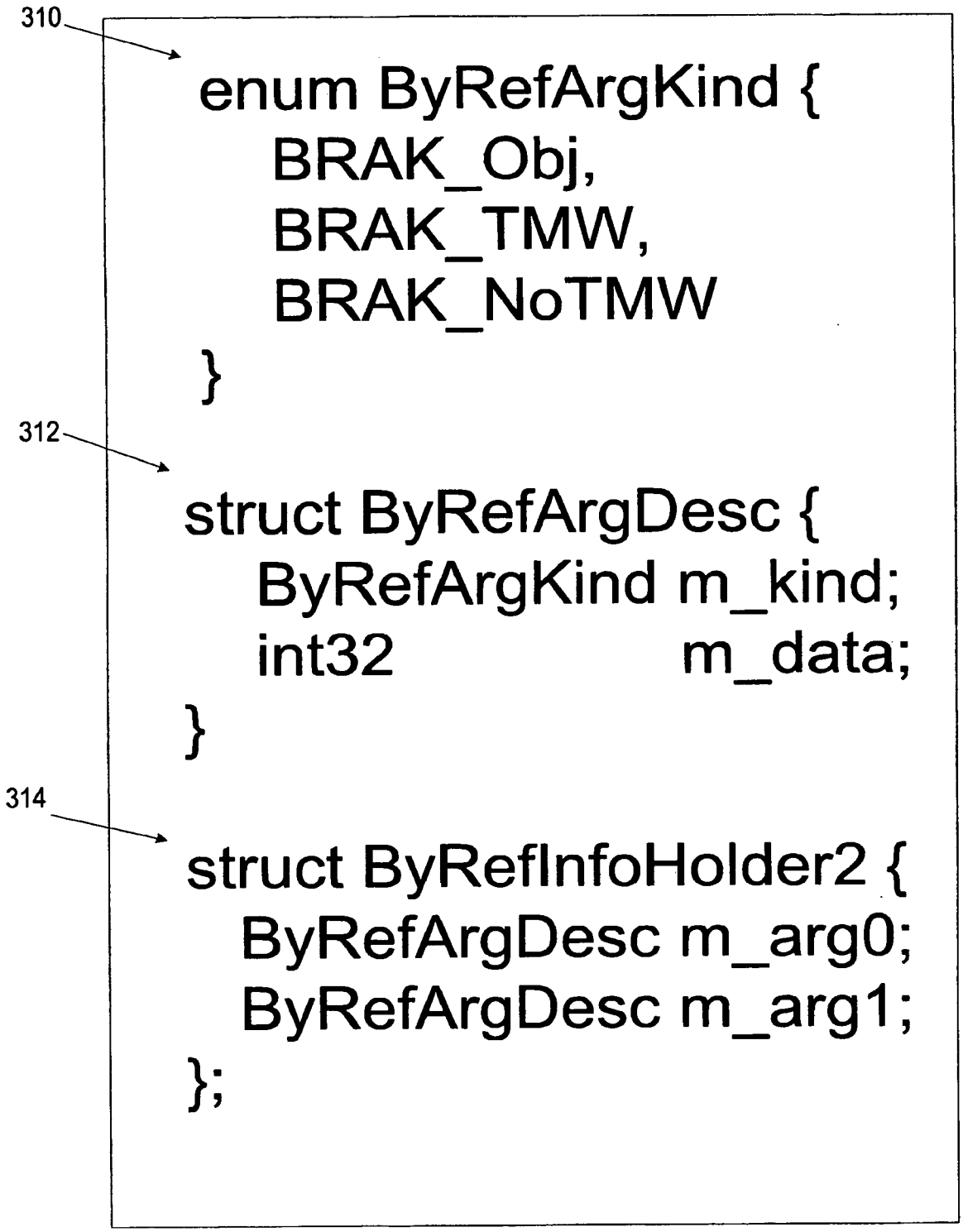

310
enum ByRefArgKind {
BRAK_Obj,
BRAK_TMW,
BRAK_NoTMW
}
312
struct ByRefArgDesc {
ByRefArgKind m_kind;
int32 m_data;
}
314
struct ByRefInfoHolder2 {
ByRefArgDesc m_arg0;
ByRefArgDesc m_arg1;
};

FIG. 8

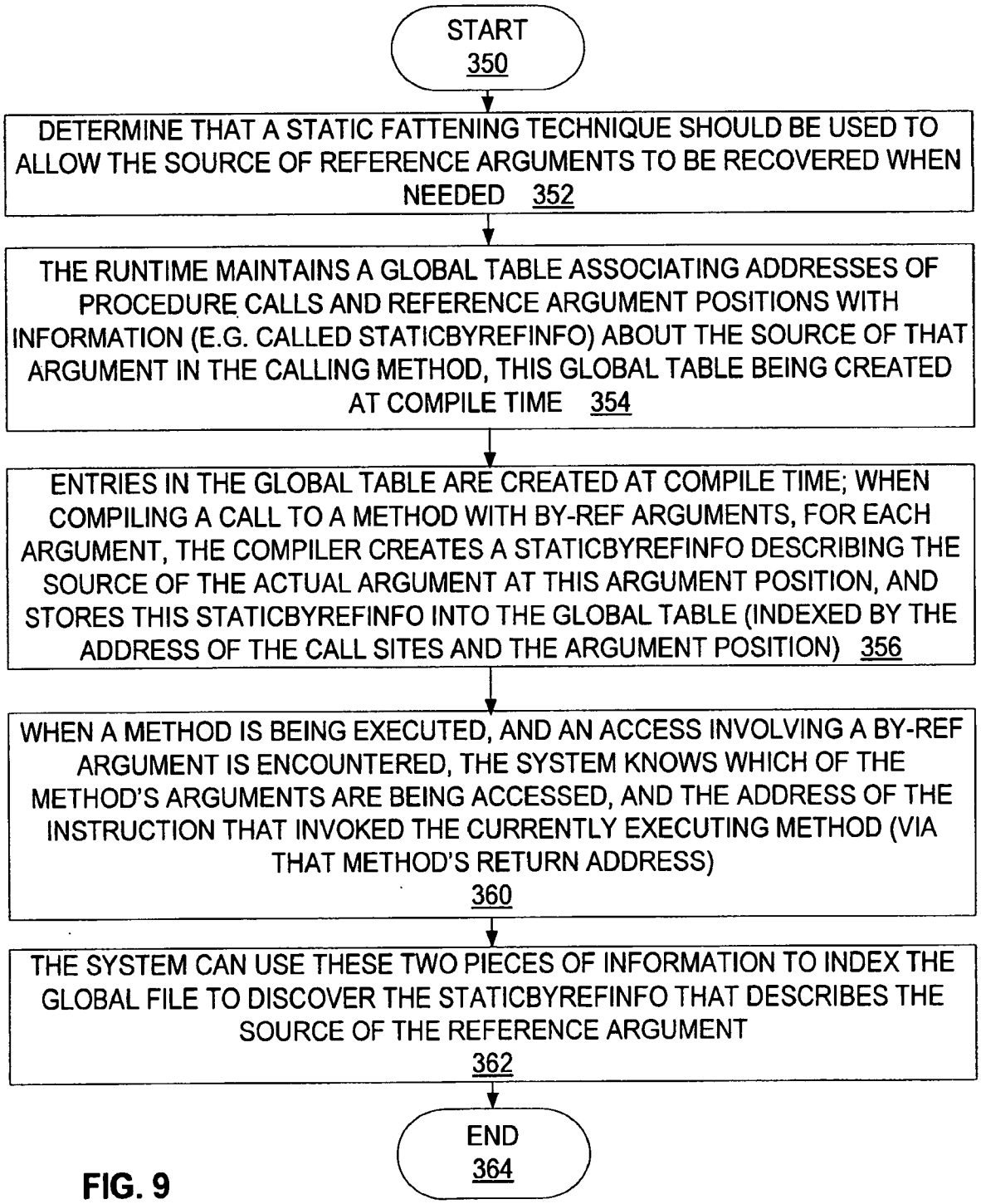

START
350
DETERMINE THAT A STATIC FATTENING TECHNIQUE SHOULD BE USED TO ALLOW THE SOURCE OF REFERENCE ARGUMENTS TO BE RECOVERED WHEN NEEDED 352
THE RUNTIME MAINTAINS A GLOBAL TABLE ASSOCIATING ADDRESSES OF PROCEDURE CALLS AND REFERENCE ARGUMENT POSITIONS WITH INFORMATION (E.G. CALLED STATICBYREFINFO) ABOUT THE SOURCE OF THAT ARGUMENT IN THE CALLING METHOD, THIS GLOBAL TABLE BEING CREATED AT COMPILE TIME 354
ENTRIES IN THE GLOBAL TABLE ARE CREATED AT COMPILE TIME; WHEN COMPILING A CALL TO A METHOD WITH BY-REF ARGUMENTS, FOR EACH ARGUMENT, THE COMPILER CREATES A STATICBYREFINFO DESCRIBING THE SOURCE OF THE ACTUAL ARGUMENT AT THIS ARGUMENT POSITION, AND STORES THIS STATICBYREFINFO INTO THE GLOBAL TABLE (INDEXED BY THE ADDRESS OF THE CALL SITES AND THE ARGUMENT POSITION) 356
WHEN A METHOD IS BEING EXECUTED, AND AN ACCESS INVOLVING A BY-REF ARGUMENT IS ENCOUNTERED, THE SYSTEM KNOWS WHICH OF THE METHOD'S ARGUMENTS ARE BEING ACCESSED, AND THE ADDRESS OF THE INSTRUCTION THAT INVOKED THE CURRENTLY EXECUTING METHOD (VIA THAT METHOD'S RETURN ADDRESS)
360
THE SYSTEM CAN USE THESE TWO PIECES OF INFORMATION TO INDEX THE GLOBAL FILE TO DISCOVER THE STATICBYREFINFO THAT DESCRIBES THE SOURCE OF THE REFERENCE ARGUMENT
362
END
364

FIG. 9

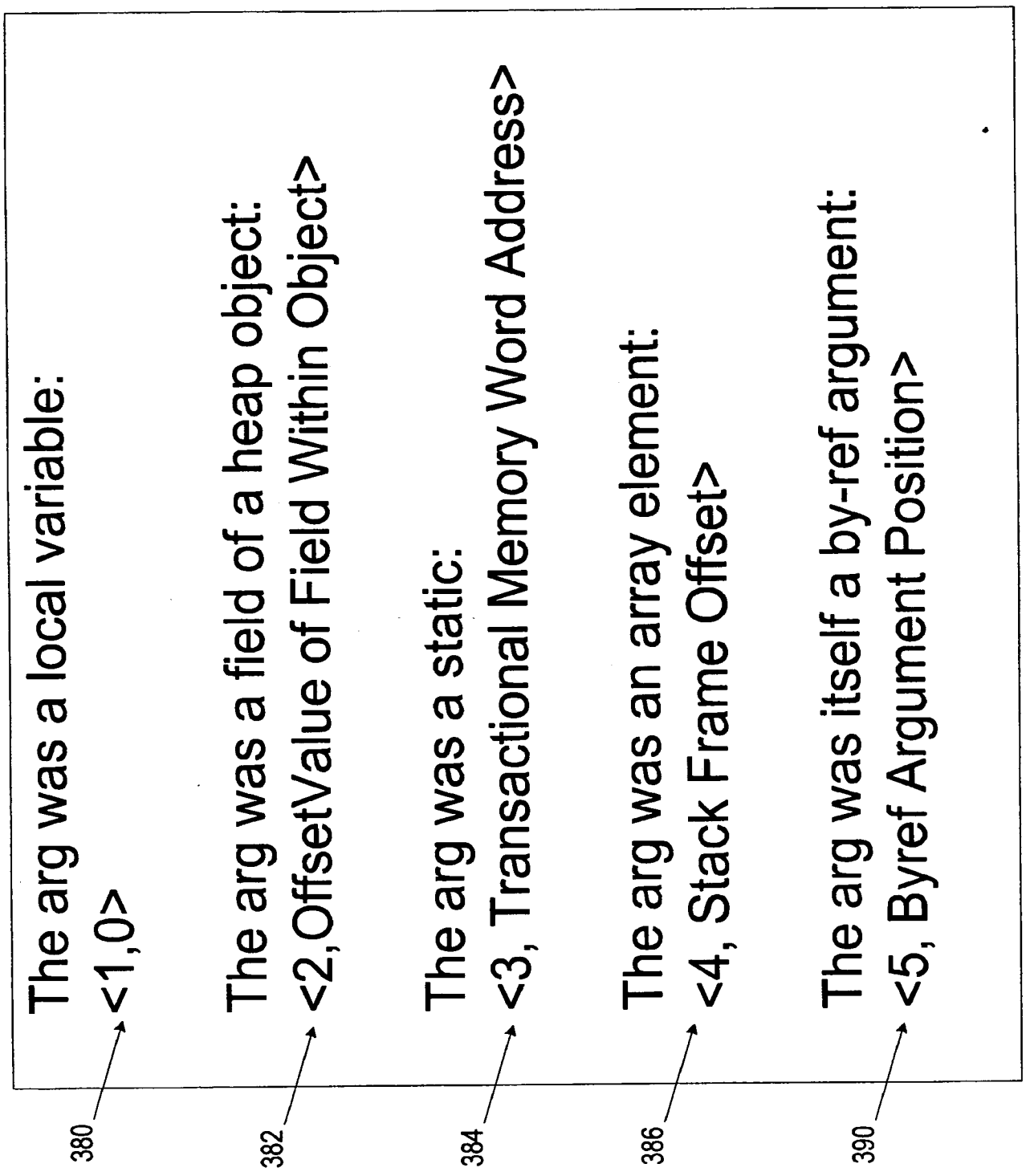

The arg was a local variable:
<1,0>
The arg was a field of a heap object:
<2,OffsetValue of Field Within Object>
The arg was a static:
<3, Transactional Memory Word Address>
The arg was an array element:
<4, Stack Frame Offset>
The arg was itself a by-ref argument:
<5, Byref Argument Position>
380
382
384
386
390

FIG. 10

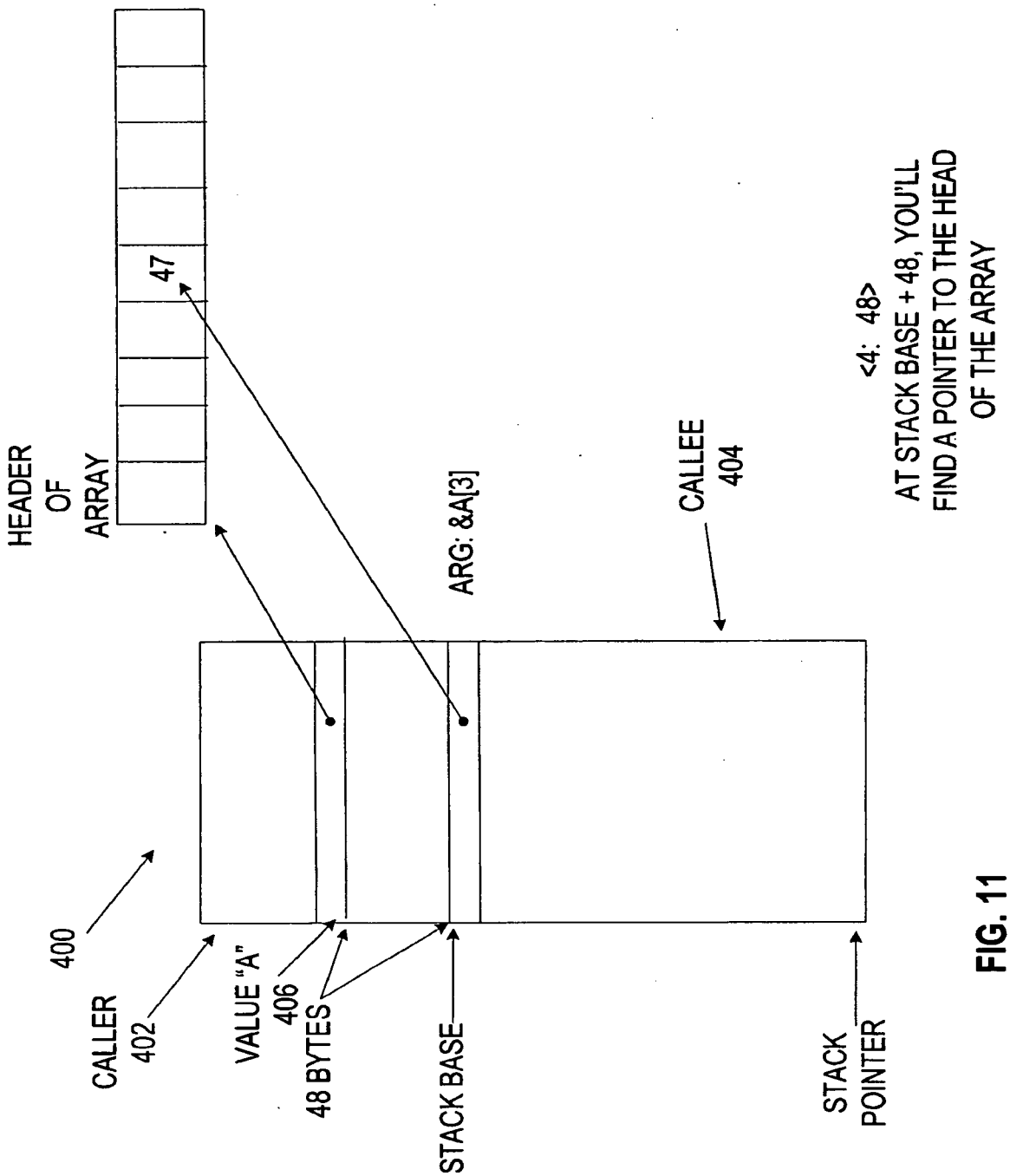

HEADER
OF
ARRAY
47
ARG: &A[3]
CALLEE
404
<4: 48>
AT STACK BASE + 48, YOU'LL
FIND A POINTER TO THE HEAD
OF THE ARRAY
400
CALLER
402
VALUE "A"
406
48 BYTES
STACK BASE
STACK
POINTER

FIG. 11

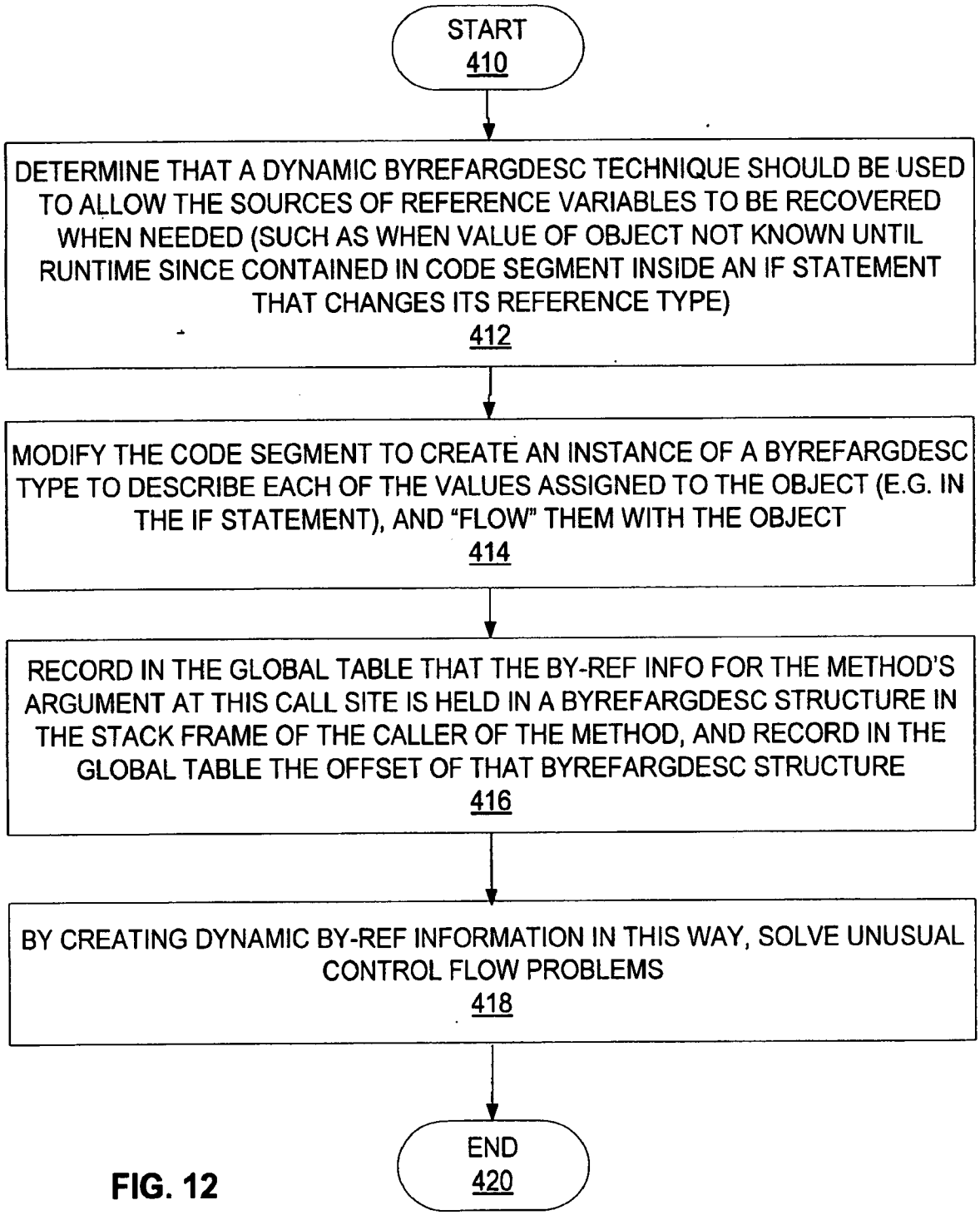

START
410
DETERMINE THAT A DYNAMIC BYREFARGDESC TECHNIQUE SHOULD BE USED TO ALLOW THE SOURCES OF REFERENCE VARIABLES TO BE RECOVERED WHEN NEEDED (SUCH AS WHEN VALUE OF OBJECT NOT KNOWN UNTIL RUNTIME SINCE CONTAINED IN CODE SEGMENT INSIDE AN IF STATEMENT THAT CHANGES ITS REFERENCE TYPE)
412
MODIFY THE CODE SEGMENT TO CREATE AN INSTANCE OF A BYREFARGDESC TYPE TO DESCRIBE EACH OF THE VALUES ASSIGNED TO THE OBJECT (E.G. IN THE IF STATEMENT), AND "FLOW" THEM WITH THE OBJECT
414
RECORD IN THE GLOBAL TABLE THAT THE BY-REF INFO FOR THE METHOD'S ARGUMENT AT THIS CALL SITE IS HELD IN A BYREFARGDESC STRUCTURE IN THE STACK FRAME OF THE CALLER OF THE METHOD, AND RECORD IN THE GLOBAL TABLE THE OFFSET OF THAT BYREFARGDESC STRUCTURE
416
BY CREATING DYNAMIC BY-REF INFORMATION IN THIS WAY, SOLVE UNUSUAL CONTROL FLOW PROBLEMS
418
END
420

FIG. 12

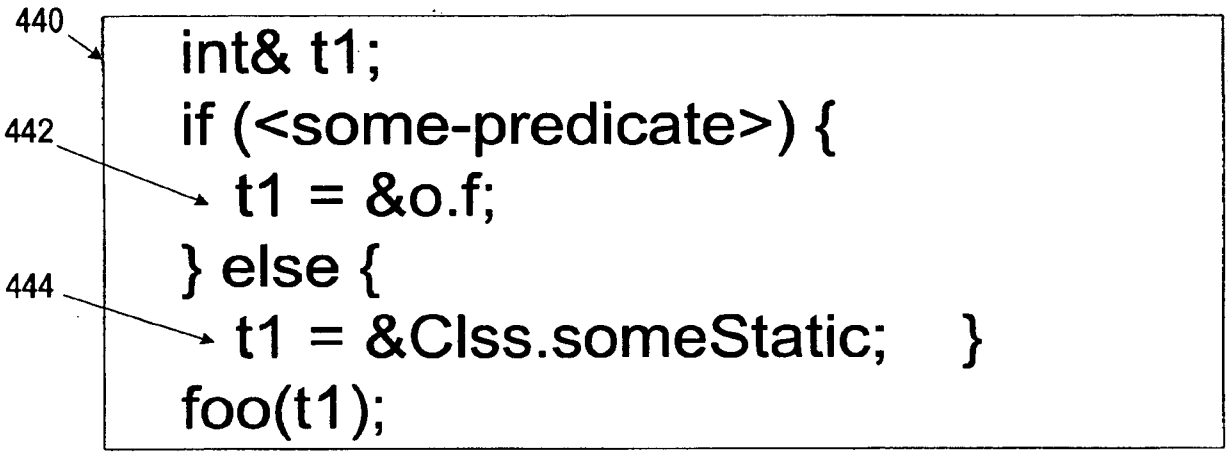

440
int& t1;
442
if (<some-predicate>) {
t1 = &o.f;
444
} else {
t1 = &Clss.someStatic; }
foo(t1);

FIG. 13

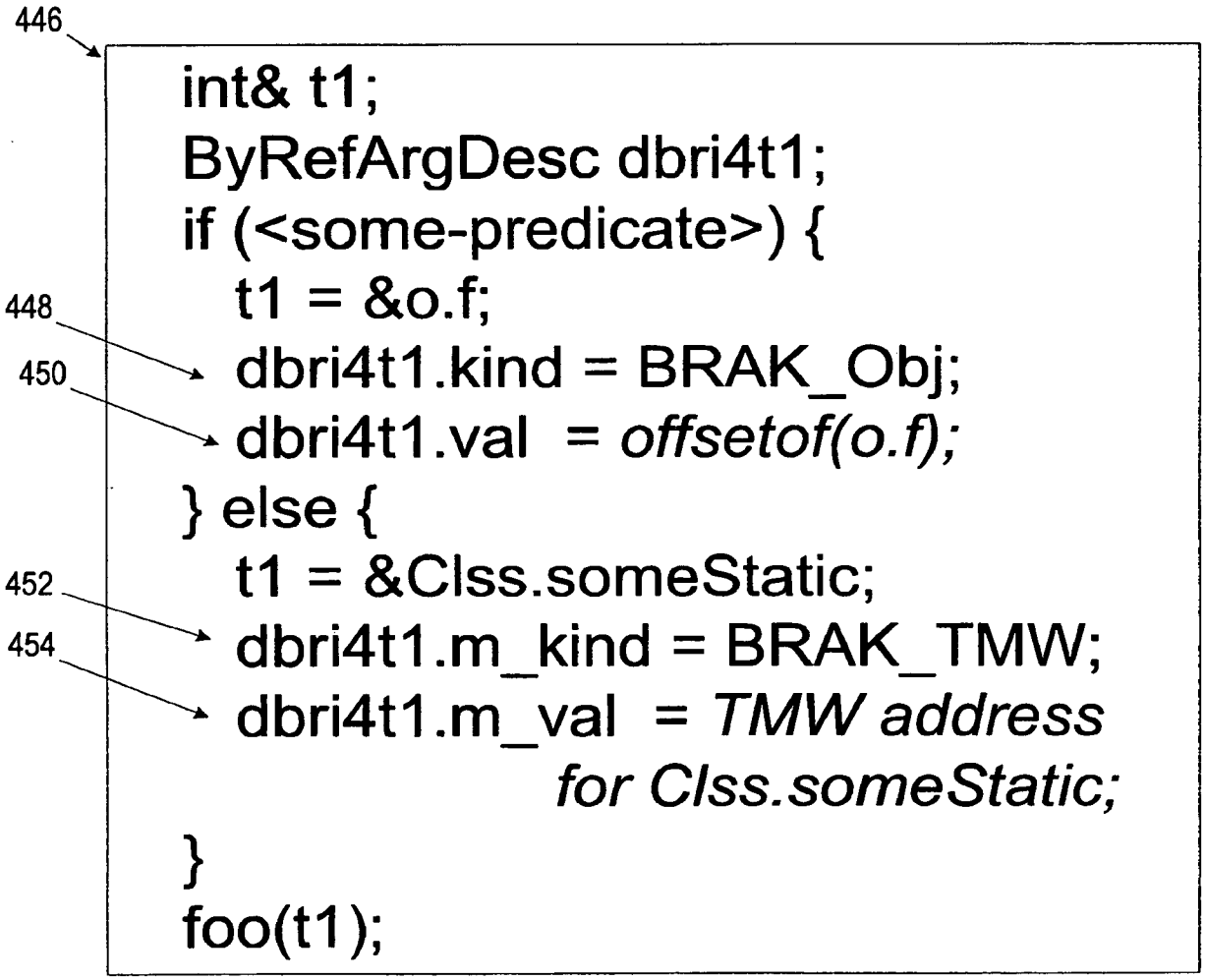

446
int& t1;
ByRefArgDesc dbri4t1;
if (<some-predicate>) {
t1 = &o.f;
448
dbri4t1.kind = BRAK_Obj;
450
dbri4t1.val = offsetof(o.f);
} else {
t1 = &Clss.someStatic;
452
dbri4t1.m_kind = BRAK_TMW;
454
dbri4t1.m_val = TMW address
for Clss.someStatic;
}
foo(t1);

FIG. 14

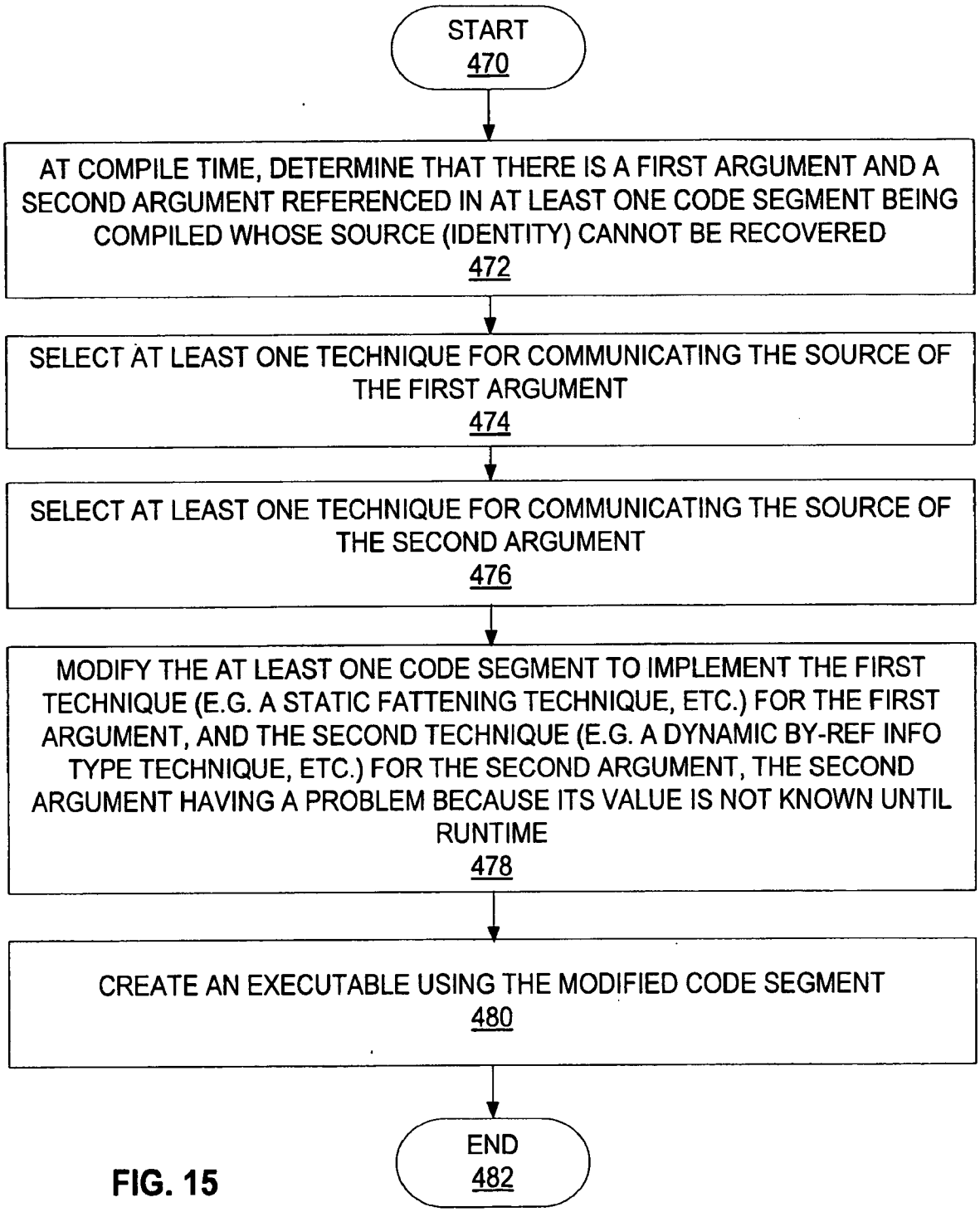

START
470
AT COMPILE TIME, DETERMINE THAT THERE IS A FIRST ARGUMENT AND A SECOND ARGUMENT REFERENCED IN AT LEAST ONE CODE SEGMENT BEING COMPILED WHOSE SOURCE (IDENTITY) CANNOT BE RECOVERED
472
SELECT AT LEAST ONE TECHNIQUE FOR COMMUNICATING THE SOURCE OF THE FIRST ARGUMENT
474
SELECT AT LEAST ONE TECHNIQUE FOR COMMUNICATING THE SOURCE OF THE SECOND ARGUMENT
476
MODIFY THE AT LEAST ONE CODE SEGMENT TO IMPLEMENT THE FIRST TECHNIQUE (E.G. A STATIC FATTENING TECHNIQUE, ETC.) FOR THE FIRST ARGUMENT, AND THE SECOND TECHNIQUE (E.G. A DYNAMIC BY-REF INFO TYPE TECHNIQUE, ETC.) FOR THE SECOND ARGUMENT, THE SECOND ARGUMENT HAVING A PROBLEM BECAUSE ITS VALUE IS NOT KNOWN UNTIL RUNTIME
478
CREATE AN EXECUTABLE USING THE MODIFIED CODE SEGMENT
480
END
482

FIG. 15

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US2007/015404

A. CLASSIFICATION OF SUBJECT MATTER

G06F 9/45(2006.01)i, G06F 9/00(2006.01)i

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)

IPC 8 : G06F

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Korean utility models and applications for utility models since 1975

Japanese utility models and applications for utility models since 1975

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)

eKIPASS(Kipo Internal), Google, YesKisti

keywords: software transactional memory, compile, call*, variable, reference, object

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
	US2004/0205740 A1 (LAVERY, D. M. et al.) 14 OCTOBER 2004	
X	See figure 5; claims 1-11.	1-15
Y	See figures 1,2A~6,7A,7B; paragraphs [26]~[30]; claim 8.	16-20
	US06718542 B1 (KOSCHE, N. et al.) 06 APRIL 2004	
X	See figure 2; claim 1.	1, 10
Y	See figure 5; column 5, lines 51-67.	16-20
A	See figures 2,5 and their descriptions; Summary.	2-9,11-15
A	US2002/0010911 A1 (CHENG, B.C. et al.) 24 JANUARY 2002 See paragraphs [25],[27].	1-20
A	US2003/0237077 A1 (GHIYA, R. et al.) 25 DECEMBER 2003 See figure 4; paragraph [28].	1-20
A	US2004/0003278 A1 (CHEN, Y. et al.) 01 JANUARY 2004 See Summary; claim 10.	1-20

☐ Further documents are listed in the continuation of Box C.

☒ See patent family annex.

* Special categories of cited documents:

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier application or patent but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents,such combination being obvious to a person skilled in the art

"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
02 JANUARY 2008 (02.01.2008)	**02 JANUARY 2008 (02.01.2008)**

Name and mailing address of the ISA/KR

Korean Intellectual Property Office
920 Dunsan-dong, Seo-gu, Daejeon 302-701,
Republic of Korea

Facsimile No. 82-42-472-7140

Authorized officer

YOON, Hye Sook

Telephone No. 82-42-481-8370

INTERNATIONAL SEARCH REPORT

Information on patent family members

International application No.

PCT/US2007/015404

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
US20040205740A1	14.10.2004	NONE	
US06718542B1	06.04.2004	NONE	
US20020010911A1	24.01.2002	NONE	
US20030237077A1	25.12.2003	NONE	
US20040003278A1	01.01.2004	NONE	