

(19) World Intellectual Property Organization
International Bureau



(43) International Publication Date
31 December 2008 (31.12.2008)

PCT

(10) International Publication Number
WO 2009/002721 A2

(51) International Patent Classification:
G06F 12/00 (2006.01)

(21) International Application Number:

PCT/US2008/066773

(22) International Filing Date: 12 June 2008 (12.06.2008)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:

11/821,630

25 June 2007 (25.06.2007) US

(71) Applicant (for all designated States except US): **MICROSOFT CORPORATION** [US/US]; One Microsoft Way, Redmond, Washington 98052-6399 (US).

(72) Inventor: **DETLEFS, David**; One Microsoft Way, Redmond, Washington 98052-6399 (US).

(81) Designated States (unless otherwise indicated, for every kind of national protection available): AE, AG, AL, AM, AO, 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, US, 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, FR, GB, GR, HR, HU, IE, IS, IT, LT, LU, LV, MC, MT, NL, NO, 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:

- without international search report and to be republished upon receipt of that report

(54) Title: **TRANSACTING ACCESSES VIA UNMANAGED POINTERS**

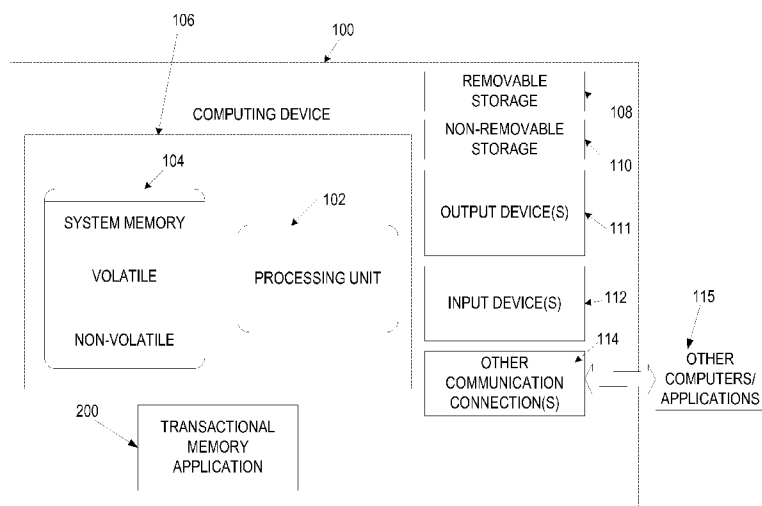


FIG. 1

(57) **Abstract:** Various technologies and techniques are disclosed for transacting accesses via unmanaged pointers in a transactional memory system. A transactional memory system is provided. Source code is analyzed to identify operations that create unmanaged pointers. Information is tracked about the targets of unmanaged pointer values in pointer variables. The target information is used to determine how accesses through an unmanaged pointer argument are to be transacted. When an unmanaged pointer is created, a descriptor of the target with the resulting pointer value is associated with the location. Within the method that creates the unmanaged pointer, the target can be identified using the descriptor, thereby enabling accesses to be transacted. When an unmanaged pointer is being passed as an argument, a descriptor is also passed as an argument to allow the unmanaged pointer target to be identified.

WO 2009/002721 A2

TRANSACTIONING ACCESSSES VIA UNMANAGED POINTERS

BACKGROUND

[001] Software transactional memory (STM) is a concurrency control mechanism analogous to database transactions for controlling access to shared memory in

5 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. STM is used as an alternative to traditional locking mechanisms. STM allows concurrent programs to be written more simply. A transaction specifies a sequence of code that is supposed to execute as if it were executing in isolation. This illusion of
10 isolation is achieved by fine-grained locking of objects, and by executing in a mode that allows the side-effects of the transaction to be rolled back if the transaction is discovered to be in conflict with some other transaction. We say that a data access is “transacted” if the code generated for that access has been modified to include support for these locking and rollback mechanisms.

15 [002] Different data accesses are transacted in different ways. For example, the lock that is acquired to synchronize access to a field is found in different ways depending on whether the field is a static (global) field or an instance field of some object. Rollback is enabled differently for local variables of methods than it is for object instance fields.

20 [003] Some languages have a “safe subset” that omits features that allow type-safety to be violated. The full language will contain some unsafe features, whose improper use may allow type-safety violations. One example of an unsafe feature is pointer arithmetic: the ability to add an integer value to a pointer value, and dereference the resulting pointer value. If the pointer is of a type that points to type
25 T, then whether the result is actually of type T depends on the correctness of the program, not on built-in guarantees of the language compiler and runtime. In the MICROSOFT® Common Language Runtime (CLR), for example, pointers may be “managed” or “unmanaged.” A managed pointer (also called a “byref”) is constant after initialization, and cannot participate in pointer arithmetic, or be cast to/from
30 other types. Unsafe code may create unmanaged pointers, which allow all these operations.

[004] This impacts transactional memory because a program may have a local variable, an instance field of some object o, and a static field, all of type “int,” and create three unmanaged pointers to these three data locations. The type of the resulting pointer is the same in all cases, “int*” (pointer to int). If the code accesses
5 memory via this pointer, we’ve lost the information necessary to transact this access in the appropriate way, which depends on the target of the pointer value.

SUMMARY

[005] Various technologies and techniques are disclosed for transacting accesses via unmanaged pointers in a software transactional memory system. A software
10 transactional memory system is provided. Source code is analyzed to identify operations that create unmanaged pointers. A static analysis tracks the targets of unmanaged pointer values as these values flow by assignments through the program. This target information is used to determine how accesses through unmanaged pointers are to be transacted.

15 [006] When an unmanaged pointer is created within a method, it may point into several different kinds of locations. These include, as discussed previously, local variables, static variables, instance variables of heap-allocated objects, or elements of heap-allocated arrays. For each of these, the static analysis associates a descriptor of the target with the resulting pointer value. A local variable requires
20 no transacting, and the descriptor for a static variable names the static variable (which is a compile-time constant, and thus can be named in a static analysis). When a pointer into heap allocated objects or array is created, the generated code is modified to store the object reference into a local variable allocated for that purpose, and the static descriptor indicates a heap-allocated object, and references
25 the corresponding local variable. Then, within the method that creates the unmanaged pointer, the target of the unmanaged pointer can be identified, thereby enabling accesses to be transacted by the unmanaged pointer.

[007] When an unmanaged pointer is being passed as an argument to a called method, a method signature is traversed to identify one or more unmanaged pointer
30 arguments. (This is combined with a similar treatment given to managed pointer, or “byref,” arguments.) At least one descriptor structure is allocated for storing a

pointer descriptor for each of the unmanaged pointer arguments. This pointer descriptor will contain a dynamic version of the static information tracked by the analysis. For locals and statics, this information is the same; for heap objects, where the static analysis indicates a local variable holding a reference to the target object, this reference is copied into the dynamic descriptor. In the called method, accesses via a pointer argument are transacted according to the information stored in the corresponding descriptor.

[008] 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

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

[010] Figure 2 is a diagrammatic view of a transactional memory application of one implementation operating on the computer system of Figure 1.

[011] Figure 3 is a high-level process flow diagram for one implementation of the system of Figure 1.

[012] Figure 4 is a process flow diagram for one implementation of the system of Figure 1 illustrating the stages involved in tracking the targets of unmanaged pointer values in pointer variables to form a tractable subset.

[013] Figure 5 is a process flow diagram for one implementation of the system of Figure 1 illustrating the stages involved in identifying intractable uses of unmanaged pointers.

[014] Figure 6 is a process flow diagram for one implementation of the system of Figure 1 illustrating the stages involved in handling unmanaged pointers that are dereferenced.

[015] Figure 7 is a process flow diagram for one implementation of the system of Figure 1 that illustrates the stages involved in passing a pointer descriptor whenever an unmanaged pointer is passed as an argument to a called method.

DETAILED DESCRIPTION

[016] 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

5 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.

[017] The system may be described in the general context as a software

10 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
15 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.

[018] In one implementation, a transactional memory system is provided that allows accesses by unmanaged pointers to be transacted safely. The term

20 “unmanaged pointer” as used herein is meant to include pointers that can be modified after they are created. A static analysis tracks the targets of unmanaged pointer values contained in pointer variables, to determine whether the pointer uses fall within a “tractable subset.” The term “tractable subset” as used herein is meant to indicate language features and usage patterns that have been identified as
25 sufficiently well behaved to allow accesses via unmanaged pointers to be transacted. The static analysis determines the target of unmanaged pointer values when they are created, and associates a descriptor of this target with the pointer value as it flows via assignment in the program. Some of these descriptors, such as those indicating that the pointer target is a local or static variable, are purely static.
30 Others have a dynamic component – when an unmanaged pointer into the interior of a heap-allocated object or array is created, a local variable is allocated and

initialized with a reference to this object or array, and the static descriptor for the pointer value references this local variable.

[019] In one implementation, new unmanaged pointer values may also be created via pointer arithmetic, by adding an integer value to a previously-created

5 unmanaged pointer value. We assume in such cases that the program uses unmanaged correctly; if a program uses unsafe features incorrectly, it may already exhibit arbitrary behavior, so failure to transact accesses correctly will make things no worse. If the program is correct, then it will only use pointer arithmetic to create pointers within the same data item that the original pointer value pointed to (or at
10 least it will only perform accesses via such pointers). Therefore, the static analysis assume that static descriptor information of an unmanaged pointer value created by adding an integer to a previously-existing unmanaged pointer value is the same as the static descriptor information for that previously-existing pointer value.

[020] Whenever an access is encountered through an unmanaged pointer, this

15 static descriptor information is used to look up information on the target of the unmanaged pointer, which allows the access to be transacted. Whenever unmanaged pointers are passed as arguments to a method call, descriptors are also passed that allows the information about the target of the unmanaged pointer to be retrieved from within the callee method. This allows accesses via the unmanaged
20 pointer in the callee method to be transacted.

[021] In some situations, insufficient information is available to allow transacting an access via a pointer. If method M1 creates an unmanaged pointer to some

target, and stores this pointer into a heap object, then later some method M2 reads this pointer and accesses its referent, M2 will not have any information about the

25 target of the pointer. We say that programs where accesses via pointers within transactions all have sufficient information to allow transacting fall within the “tractable subset” of the languages; programs like the one above, where there is insufficient information to allow transacting, fall outside the tractable subset. In cases like this, the programmer can modify the program to use a “tracked pointer”
30 instead of a raw pointer; this idea will be discussed more fully below.

[022] 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.

[023] 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 be accessed by device 100. Any such computer storage media may be part of device 100.

[024] 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 transactional memory application

200. Transactional memory application 200 will be described in further detail in Figure 2.

[025] Turning now to Figure 2 with continued reference to Figure 1, transactional memory application 200 operating on computing device 100 is illustrated.

5 Transactional memory application 200 is one of the application programs that reside on computing device 100. However, it will be understood that transactional memory application 200 can alternatively or additionally be embodied as computer-executable instructions on one or more computers and/or in different variations than shown on Figure 1. Alternatively or additionally, one or more parts of
10 transactional memory application 200 can be part of system memory 104, on other computers and/or applications 115, or other such variations as would occur to one in the computer software art.

[026] Transactional memory application 200 includes program logic 204, which is responsible for carrying out some or all of the techniques described herein.

15 Program logic 204 includes logic for providing a transactional memory system 206; logic for analyzing source code to identify operations that create unmanaged pointers from object or other addresses 208; logic for tracking information about the targets of unmanaged pointer values in pointer variables 210; logic for statically identifying intractable uses of unmanaged pointers within transactions, and
20 reporting such uses as errors 212; logic for allowing unmanaged pointer values to be created by pointer arithmetic 213; logic for passing a descriptor any time an unmanaged pointer is passed as an argument to a called method (e.g. the descriptor describing the kind of location the pointer targets, and other information necessary to identify the target, such as the address of the object into which the pointer points)
25 214; logic for using the target information to determine how accesses through an unmanaged pointer is to be transacted 216; and other logic for operating the application 220. In one implementation, program logic 204 is operable to be called programmatically from another program, such as using a single call to a procedure in program logic 204.

30 [027] Turning now to Figures 3-7 with continued reference to Figures 1-2, the stages for implementing one or more implementations of transactional memory

application 200 are described in further detail. Figure 3 is a high level process flow diagram for transactional memory application 200. In one form, the process of Figure 3 is at least partially implemented in the operating logic of computing device 100. The process begins at start point 240 with providing a transactional memory system (stage 242). The system then analyzes the source code (e.g. intermediate language or original source code) to identify operations that create unmanaged pointers from object or other addresses (stage 244).

[028] A static analysis tracks information about the targets of unmanaged pointer values in pointer variables (e.g. tracks the object it is a pointer into and what by-ref, objref, and unmanaged pointer values the local variables contain) (stage 246).

Unmanaged pointers values may be created by pointer arithmetic, by adding an integer value to a previously-existing unmanaged pointer value (stage 247). In this case, we assume that the target information of the resulting pointer value is the same as that of the previously-existing pointer value. This reflects the assumption that the program uses unsafe features correctly, and never accesses outside the bounds of the original data item into which an unmanaged pointer points. Any intractable uses of unmanaged pointers are identified and reported as errors (stage 248). Whenever an unmanaged pointer is being passed as an argument to a called method, a descriptor is also passed that describes the kind of location the pointer targets, and other information necessary to identify the target (stage 250). The system uses the target information to determine accesses through unmanaged pointers are to be transacted (stage 252). The process ends at end point 254.

[029] Not all allowed uses of unmanaged pointers can be tracked via a static analysis of this kind. For example, one part of a program may create an unmanaged pointer, and store it into a field of a heap object. Another part of the program may read this pointer value from the heap object, and then perform an access via this pointer. Statically, we don't have information that enables transacting of this access. We say that such non-transactable accesses fall outside of the "tractable subset" of unsafe code; the accesses we can transact via this static analysis are within the tractable subset. In cases like this, the programmer can modify the program to use a "tracked pointer" instead of a raw pointer. In one

implementation, a tracked pointer will contain both a pointer value and its associated dynamic target information; when a raw pointer value is assigned to a tracked pointer, the static descriptor information on the raw pointer will be assigned to the dynamic tracked pointer, and this information will be retrieved and used
5 when the pointer value is read from the tracked pointer.

[030] Figure 4 illustrates one implementation of the stages involved in tracking the sources of unmanaged pointer values in pointer variables. In one form, the process of Figure 4 is at least partially implemented in the operating logic of computing device 100. The process begins at start point 270. When creating an unmanaged
10 pointer within a method, a descriptor of the target is associated with the resulting pointer value (stage 272). If the target is a local variable (decision point 273), the local variable requires no transacting, so the descriptor for a pointer to a local variable indicates this (stage 274). If the target is a static variable (decision point 275), the descriptor for a static variable actually names the variable (stage 276). If
15 the target is not a local variable or a static variable, then the target is a heap allocated object or array (stage 278). In such a case, the generated code is modified to store the object or array references in a local variable allocated for that purpose (stage 280). Furthermore, the static descriptor indicates a heap-allocated object, and references the corresponding local variable (stage 282). As discussed
20 previously, unmanaged pointer values created by pointer arithmetic from previously-existing unmanaged pointer values inherit the static descriptor of the previous pointer value, on the assumption that a correct program will not access via an "out-of-bounds" pointer value. In any event, within the method that creates the unmanaged pointer, the target can be identified using the descriptor which allows
25 accesses via the unmanaged pointer to be transacted (stage 284). The process ends at end point 286.

[031] Figure 5 illustrates one implementation of the stages involved in identifying intractable uses of unmanaged pointers. In one form, the process of Figure 5 is at least partially implemented in the operating logic of computing device 100. The
30 process begins at start point 290 with analyzing the source code to identify operations that create unmanaged pointers from object or other addresses (stage

292). The system determines if any of the identified unmanaged pointers are intractable (e.g. because it accesses a location via an unmanaged pointer that is not being tracked because it was not created locally) (stage 294). Accesses via such intractable pointers are intractable (stage 296). In some cases, these can be identified statically, and can be immediately reported to users as errors. In other cases, they may only be identified dynamically. For example, a method M1 may pass an intractable pointer value to a called method M2. This is an error only if M2 perform an access via this pointer value. In a separation compilation environment, when M1 is compiled we may not know whether M2 will perform an access via the pointer, and when M2 is compiled we may not know that any caller will pass an intractable pointer. The process ends at end point 298.

[032] Figure 6 illustrates one implementation of the stages involved in handling unmanaged pointers that are dereferenced. In one form, the process of Figure 6 is at least partially implemented in the operating logic of computing device 100. The process begins at start point 310 with determining that an unmanaged pointer has been dereferenced (stage 312). If the pointer was tracked as part of the tractable subset (decision point 314), then the dereferencing is fine (stage 316). However, if the pointer was not tracked as part of the tractable subset (decision point 314), then an exception is raised (stage 318) or other appropriate action taken. The process ends at end point 320.

[033] Figure 7 illustrates one implementation of the stages involved in passing a pointer descriptor whenever an unmanaged pointer is passed as an argument to a called method. In one form, the process of Figure 7 is at least partially implemented in the operating logic of computing device 100. The process begins at start point 340 by identifying that an unmanaged pointer is being passed as an argument to a called method (stage 342). If the unmanaged pointer is not part of the tractable subset (decision point 344), then an error may be raised or other appropriate action may be taken (stage 354) and the process may end at end point 356. Otherwise, the unmanaged pointer may be passed to the called method, which will raise an error if the intractable pointer is derereferenced. If the unmanaged pointer is part of the tractable subset (decision point 344), the method signature is

traversed to identify unmanaged pointer arguments (stage 346). The caller then allocates a descriptor structure, with one descriptor for each unmanaged pointer argument (e.g. a dynamic version of the static information tracked by the analysis) (stage 348). For locals and statics, the dynamic version is the same as the static information (stage 350). For heap objects, where the static analysis indicates a local variable holding a reference to the target object, this reference is copied into the dynamic descriptor (stage 351). When the caller finds an access via an unmanaged pointer argument, the dynamic object descriptor is used to allow the object to be transacted (e.g. for heap objects it contains a reference to the head of the heap object) (stage 352). The process ends at end point 356.

[034] 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.

[035] For example, a person of ordinary skill in the computer software art will recognize that the client and/or server arrangements, user interface screen content, 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 computer-readable medium having computer-executable instructions for causing a computer to perform steps comprising:
 - provide a transactional memory system (200);
 - 5 analyze source code to identify operations that create unmanaged pointers (208);
 - provide a static analysis that tracks targets of unmanaged pointer values (210); and
 - use the targets to determine how accesses through unmanaged pointers are to
 - 10 be transacted (216).
2. The computer-readable medium of claim 1, further comprising:
 - identify intractable uses of the unmanaged pointers (212); and
 - indicating errors when such intractable uses occur within a transaction (212).
3. The computer-readable medium of claim 1, wherein the static analysis tracks
- 15 targets of unmanaged pointer values as the unmanaged pointer values flow by assignments through the transactional memory system (246).
4. The computer-readable medium of claim 1, wherein the static analysis tracks targets of unmanaged pointer values that are created by pointer arithmetic from previously existing unmanaged pointer values (247).
- 20 5. The computer-readable medium of claim 1, wherein an intermediate language version of the source code or an original source code is analyzed to identify the operations that create unmanaged pointers (244).
6. The computer-readable medium of claim 1, further comprising:
 - when an unmanaged pointer is being passed to a called method, also pass a
 - 25 descriptor that allows a target of the unmanaged pointer to be reconstructed in the called method (250).
7. The computer-readable medium of claim 6, wherein the descriptor is a dynamic version of static information tracked by the static analysis (350).
8. The computer-readable medium of claim 1, wherein the unmanaged pointers
- 30 are created from object or other addresses (244).

9. A method for tracking targets of unmanaged pointer values comprising the steps of:

providing a transactional memory system (242);

when creating an unmanaged pointer within a method, associating a
5 respective descriptor of a target into which the unmanaged pointer points with the unmanaged pointer (272); and

within the method that creates the unmanaged pointer, allowing the target on which the unmanaged pointer is based to be identified using the respective descriptor, thereby enabling accesses to be transacted by the unmanaged pointer
10 (284).

10. The method of claim 9, wherein if the unmanaged pointer points to a location that is a local variable, then the respective descriptor will indicate that accesses via the pointer do not need to be transacted (284).

11. The method of claim 9, wherein if the unmanaged pointer points to a
15 location that is a static variable, then the respective descriptor will actually name the static variable (276).

12. The method of claim 9, wherein any locations that are for heap allocated objects will have a local variable allocated to store a respective object reference, and will have the respective descriptor reference the local variable (282).

20 13. A computer-readable medium having computer-executable instructions for causing a computer to perform the steps recited in claim 9 (200).

14. A method for passing a pointer descriptor whenever an unmanaged pointer is passed as an argument comprising the steps of:

providing a transactional memory system (242);

25 identifying that an unmanaged pointer is being passed as an argument to a called method (342);

traversing a method signature to identify one or more unmanaged pointer arguments (346); and

allocating at least one descriptor structure for storing a pointer descriptor for
30 each of the unmanaged pointer arguments (348).

15. The method of claim 14, wherein the traversing and allocating steps are only performed if accesses via the unmanaged pointer would be tractable (344).

16. The method of claim 14, wherein the traversing and allocating steps are not performed if accesses via the unmanaged pointer would not be tractable, and an
5 error is instead raised (354).

17. The method of claim 14, wherein when a caller finds an access via the unmanaged pointer argument, a respective pointer descriptor in the descriptor structure is used to allow an object to be transacted (352).

18. The method of claim 17, wherein the respective pointer descriptor contains a
10 reference to the heap object into which the unmanaged pointer points (351).

19. The method of claim 14, wherein the descriptor structure is allocated by a caller method (348).

20. A computer-readable medium having computer-executable instructions for causing a computer to perform the steps recited in claim 14 (200).

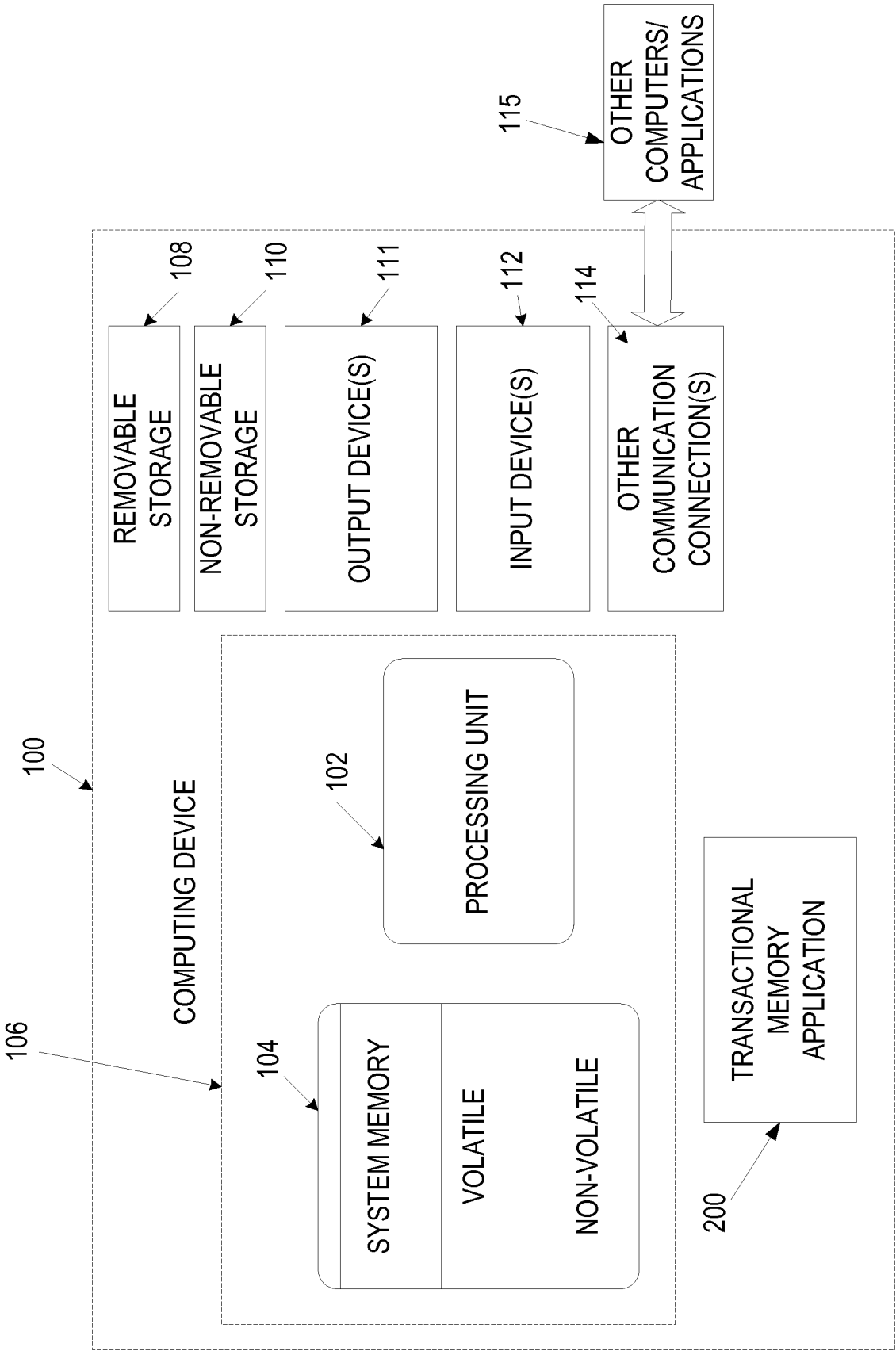


FIG. 1

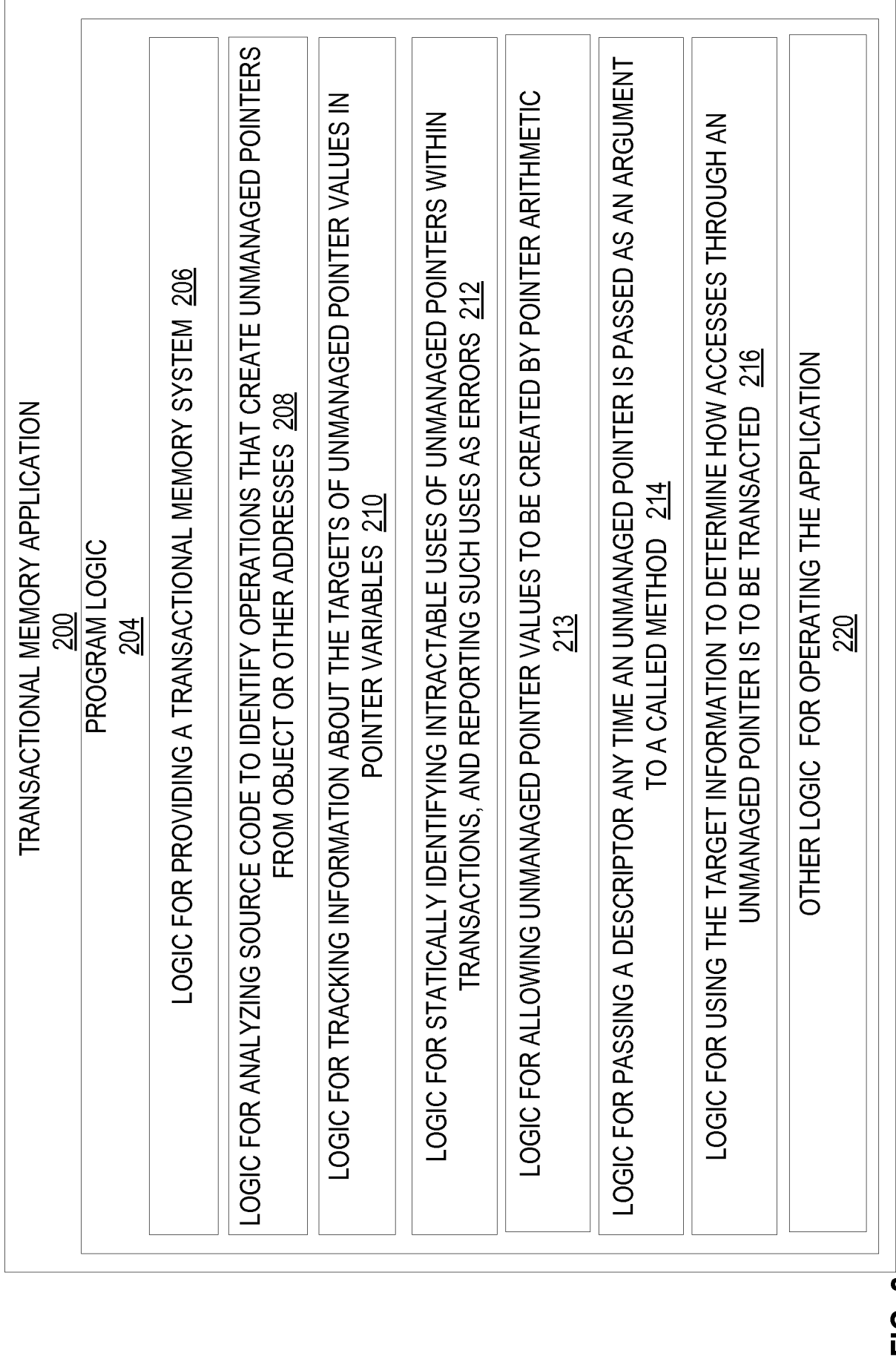


FIG. 2

3 / 7

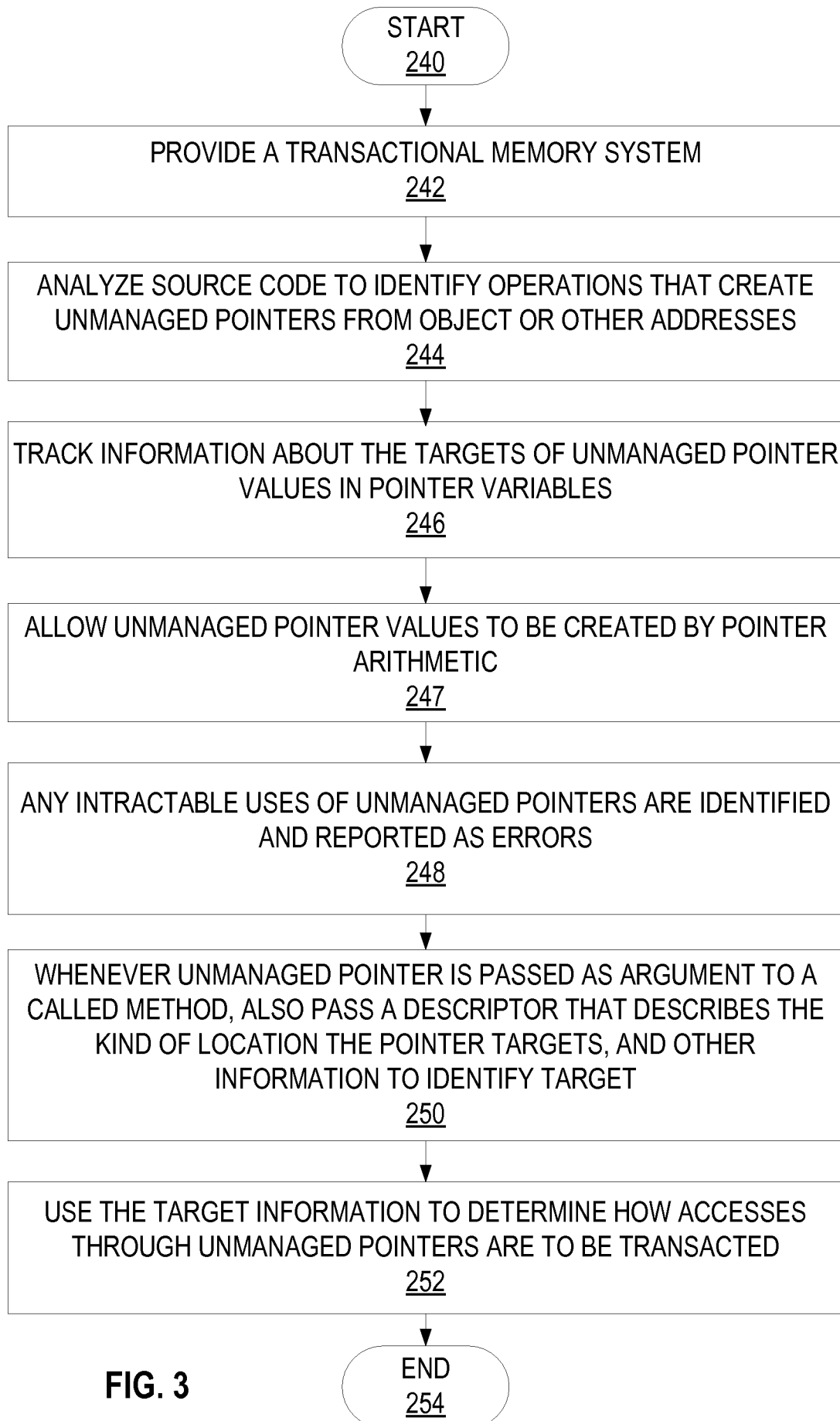
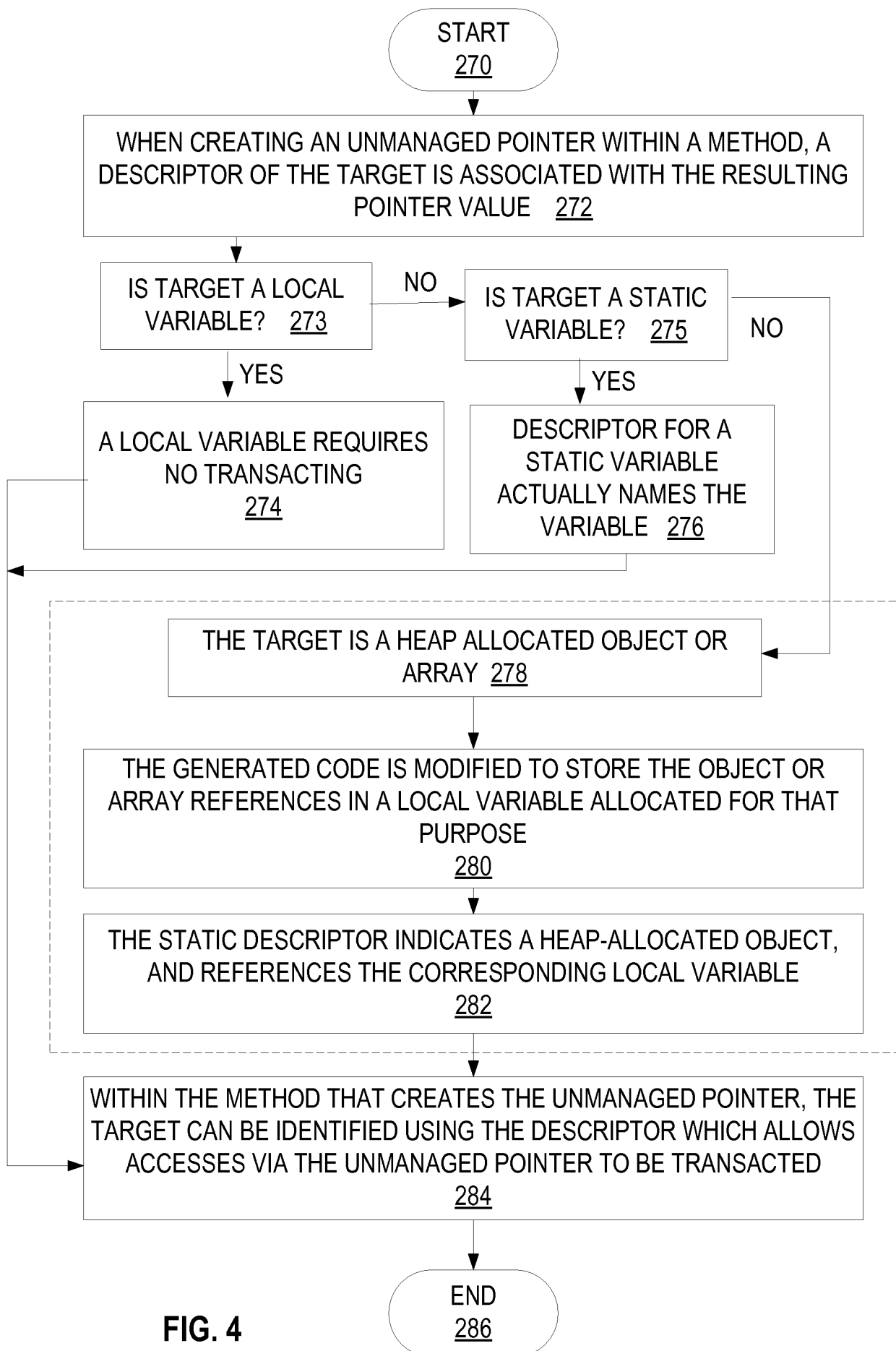


FIG. 3

4 / 7



5 / 7

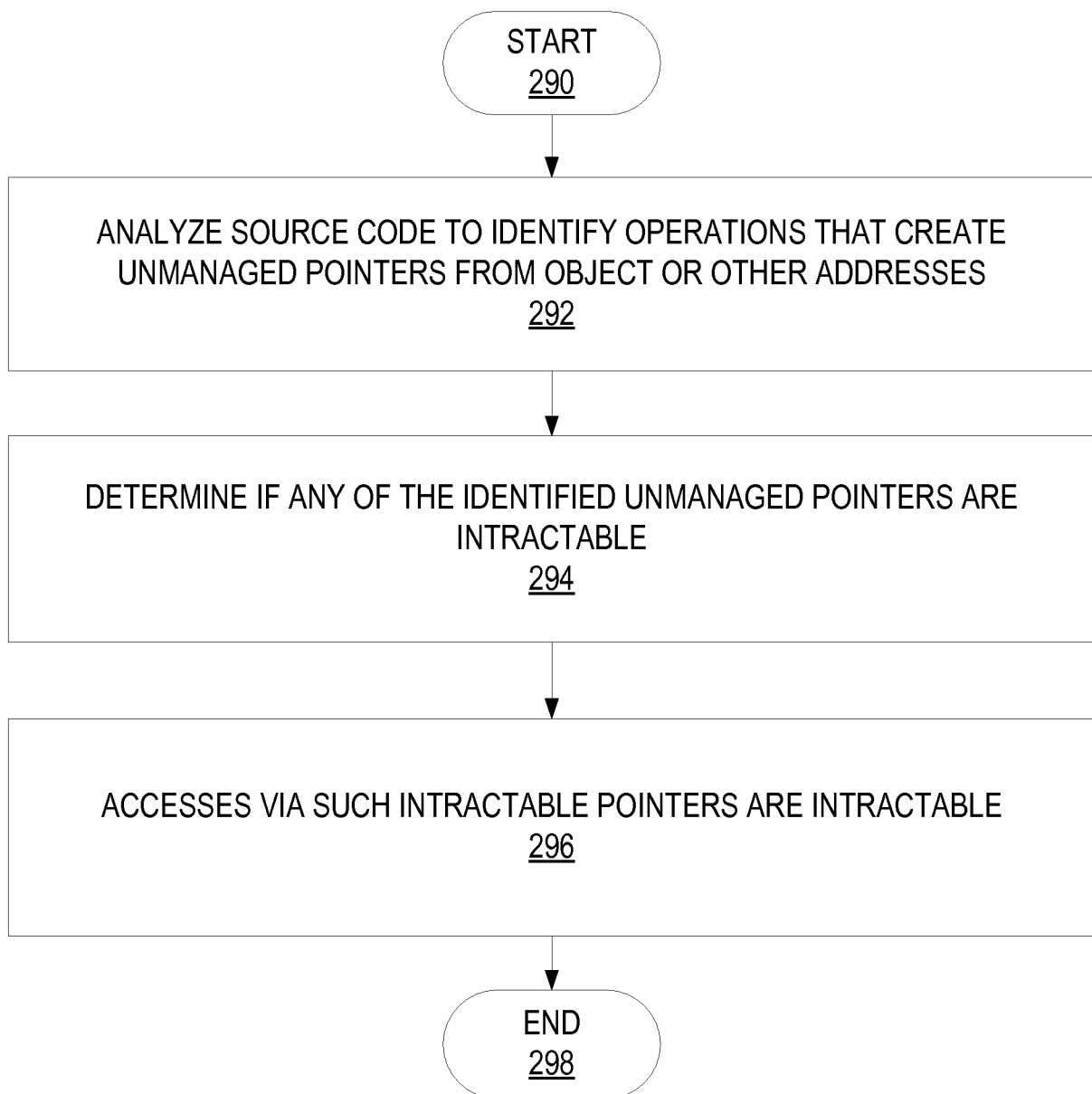
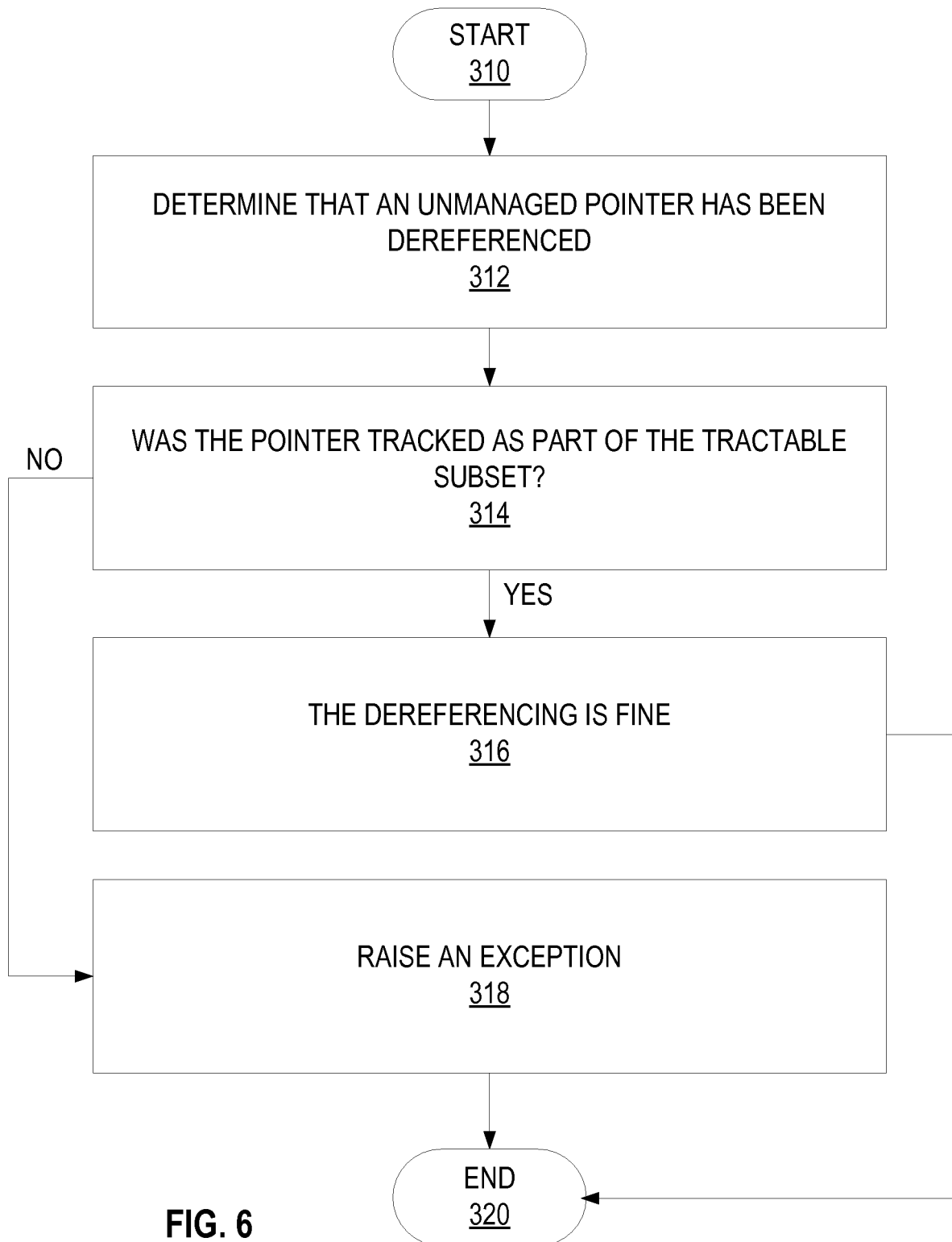


FIG. 5

6 / 7



7 / 7

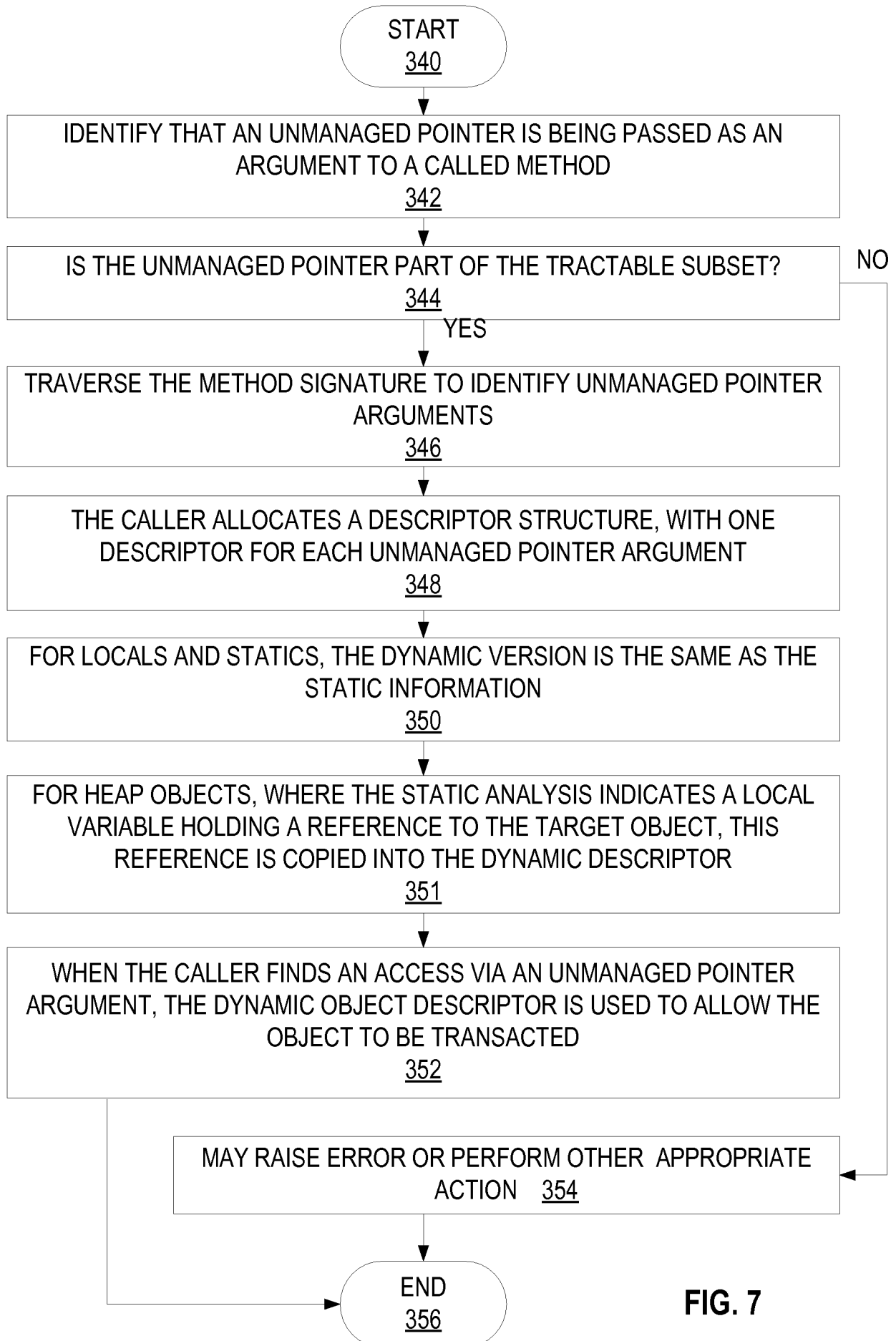


FIG. 7