(19) **Bundesrepublik Deutschland**
Deutsches Patent- und Markenamt

(10) **DE 698 26 700 T2** 2005.10.20

(12) # Übersetzung der europäischen Patentschrift

(97) **EP 1 025 491 B1**
(21) Deutsches Aktenzeichen: **698 26 700.1**
(86) PCT-Aktenzeichen: **PCT/US98/22261**
(96) Europäisches Aktenzeichen: **98 953 827.7**
(87) PCT-Veröffentlichungs-Nr.: **WO 99/021085**
(86) PCT-Anmeldetag: **21.10.1998**
(87) Veröffentlichungstag der PCT-Anmeldung: **29.04.1999**
(97) Erstveröffentlichung durch das EPA: **09.08.2000**
(97) Veröffentlichungstag der Patenterteilung beim EPA: **29.09.2004**
(47) Veröffentlichungstag im Patentblatt: **20.10.2005**

(51) Int Cl.[7]: **G06F 9/44**

(30) Unionspriorität:
954843 **21.10.1997** **US**

(84) Benannte Vertragsstaaten:
DE, GB

(73) Patentinhaber:
FTL Systems Inc., Rochester, Minn., US

(72) Erfinder:
WILLIS, C., John, Rochester, US; NEWSHUTZ, N., Robert, Rochester, US

(74) Vertreter:
BOEHMERT & BOEHMERT, 28209 Bremen

(54) Bezeichnung: **KOMPILERORIENTIERTES GERÄT ZUR PARALLELKOMPILATION, SIMULATION UND AUSFÜHRUNG VON RECHNERPROGRAMMEN UND HARDWAREMODELLEN**

Beschreibung

Hintergrund der Erfindung

[0001] Das Verteilen der Kompilierung, Simulation und Ausführung von Computerprogrammen und Hardwaremodellen auf zwei oder mehr Verarbeitungsknoten hat zwei primäre Vorteile: eine gesteigerte Programm-/Modellkapazität und eine verringerte Simulations-/Ausführungszeit. Die Größe und Komplexität eines Programms-/Modells, das kompiliert und simuliert/ausgeführt werden kann, steigt auf Grund des zusätzlichen Speichers als auch der verfügbaren Verarbeitungsressourcen. Die Simulations-/Ausführungszeit nimmt auf Grund der Möglichkeit von Zugriffen für die Optimierung auf den teilweise kompilierten Hilfszustand, der das Programm/Modell repräsentiert, als auch auf Grund gleichzeitiger Kompilierung, Simulation und Ausführung mit Hilfe von Mehrfachverarbeitungsknoten ab.

[0002] Ein Verarbeitungsknoten besteht aus einem oder mehreren Allzweckprozessoren, die einen gemeinsamen Speicher teilen. Optionale Bestandteile eines Verarbeitungsknotens umfassen prozessorspezifischen Speicher, verschiedene Ebenen von Cache-Speicherung, die für einen einzelnen Prozessor spezifisch sind oder von zwei oder mehr Prozessoren geteilt werden, und rekonfigurierbare Logik, die für einen einzelnen Prozessor spezifisch oder die zwei oder mehr Prozessoren gemeinsam ist. Verarbeitungsknoten können einen oder mehrere getrennte virtuelle Adressräume unterstützen, die auf physikalische Speichervorrichtungen mittels herkömmliche Adressübersetzungshardware und -software abgebildet werden. Die Verarbeitungsknoten können als Mehrfachprozessoren mit gemeinsamem Speicher angesehen werden, zu denen rekonfigurierbare Logikanordnungen zugefügt worden sind.

[0003] Verarbeitungsknoten (und Mehrfachprozessoren mit gemeinsamem Speicher) werden bereits in Konfigurationen hergestellt, die bis zu ungefähr einem Dutzend Prozessoren enthalten. Wenn jedoch zusätzliche Prozessoren mit einer Verbindung zu einem gemeinsamen geteilten Speicher zugefügt werden, wird die Effizienz von jedem Prozessor auf Grund der Verbindung zu dem gemeinsamen, geteilten Speicher verschlechtert. Daher werden größere und leistungsfähigere Rechensysteme häufig geschaffen, indem zwei oder mehr solcher Verarbeitungsknoten verbunden werden, in dem Punkt-zu-Punkt- (point-to-point-) oder Mehrfachausstrahlungs-Nachrichten-Protokolle (multi-cast message protocols) verwenden werden. Punkt-zu-Punkt-Nachrichten-Protokolle kommunizieren eine Einheit von Informationen (eine Nachricht) von einem Agenten auf einem Verarbeitungsknoten zu einem Agenten auf dem gleichen Verarbeitungsknoten oder einem anderen Verarbeitungsknoten. Mehrfachausstrahlungs-Nachrichten-Protokolle kommunizieren von einem Agenten auf einem Verarbeitungsknoten mit einem oder mehreren Agenten auf dem gleichen oder anderen Verarbeitungsknoten. Agentenfunktionalität ist entweder als Software, die auf Prozessoren ausgeführt wird, oder als Hardware ausgeführt, die in rekonfigurierbare Logikanordnungen eingebettet oder mit diesen verknüpft ist. Solche Agenten verkörpern Bestandteile der Kompilierung, der Simulation oder der Ausführung.

[0004] Die Kompilierung, die Simulation und die Ausführung werden produktiv als eng miteinander verknüpfte Betriebsarten angesehen, die in Prozessorausführbarem (manifestiert in Cache-Speichern und einem Speicher) und einer Logikkonfiguration (manifestiert in rekonfigurierbaren Logikelementen) verkörpert sind. Die Kompilierung übersetzt ein oder mehrere Computerprogramme und/oder Hardwaremodelle in Prozessorausführbares und Logikkonfigurationsinformationen. Das Verhalten, das von dem Ausführbaren und der Logikkonfiguration repräsentiert wird, kann dann als Simulation bewertet werden und/oder ausgeführt werden. Im allgemeinen Gebrauch bezieht sich Simulation häufig auf die Berechnung von Hardwaremodellen, wohingegen die Ausführung sich oft auf die Berechnung eines Computerprogramms bezieht. Mit der zunehmenden Verwendung von Hardwarebeschreibungssprachen (z. B. VHDL und Verflog) als auch Hardware-/Software-Mit-Design (siehe EP-A-0772140) sind die Simulation und die Ausführung fast ununterscheidbare Betriebsarten geworden und werden im folgenden als solche behandelt.

[0005] Um inkrementale Betriebsarten in Programme oder Modelle aufzunehmen, zum Beispiel als symbolisches Testen (debugging), Profilieren, Fehlereinfügen, selektives Ereignisverfolgen, dynamisches Verknüpfen von Bibliotheken, inkrementelles Optimieren von Ausführbarem (basierend auf verfügbaren Ressourcen oder neuen Informationen) und von Programmierungsschnittstellen, die nach der inkrementellen Modifizierung des Programms/Modells während der Ausführung/Simulation verlangen, ist es für die Kompilierungs- und Ausführungs-/Simulationsarten nützlich, eng gekoppelt zu sein. Solch eine enge Kopplung verringert die Simulations-/Ausführungszeit bei gegebenen festen Ausführungsressourcen.

[0006] Die Kompilierung wird typischerweise in einer unidirektionalen Pipeline angeordnet, die zwei oder mehr Hilfsdateien (tatsächliche oder im Speicher über Röhren (pipes) simulierte) verwendet, bevor die Ausführungs-/Simulationsbetriebsart erreicht wird. Gemeinsame Hilfsdateien umfassen Hilfsoptimierungsrepräsentationen, Textzusammenbaucode (textual assembly code), verschiebbare Binärdaten (binaries) und ausführbare Dateien. Viele Simulato-

ren führen sogar eine Hilfsprogrammiersprache ein, wenn die Kompilierung eines Hardwaremodells in ein Programm übersetzt wird, welches dann von einem programmiersprachenspezifischen Kompilieren kompiliert wird. Einige optimierende Kompilieren nutzen bis zu einem Dutzend Dateihilfsglieder.

[0007] Die Verwendung von Vorrichtungen, wie Dateien, um in einer Richtung zwischen Phasen der Kompilierung zu kommunizieren, verhindert den schnellen und effizienten Fluß von Informationen zurück von den hinteren Stufen zu den früheren Stufen der Kompilierungsbetriebsart. Zum Beispiel können eine Back-End-Kompilierfunktionalität, die prozessorausführbare Anweisungen in dem gemeinsamen Speicher anordnet, oder eine Logikfunktionalität innerhalb der rekonfigurierbaren Logikanordnungen ein falsches gemeinsames Verwenden einer rekonfigurierbaren Logik-Stift-Verbindung feststellt, der man sich am effizientesten zuwendet, indem frühere Kompilierungsfunktionalität (in diesem Fall das Abbilden und das Planen) teilweise erneut ausführt wird, um eine optimalere Simulations-/Ausführungsauslastung zu erzeugen.

[0008] Dateien stellen ferner einen sehr groben Kommunikationsmechanismus zwischen Stufen der Kompilierung dar. Wesentliche Information liegen allgemein in einer Dateihilfsstufe vor, die für eine örtliche Änderung der Ausführung/Simulation irrelevant ist. Solch eine Kompilierung oder erneute Kompilierung muß wesentlich mehr Informationen handhaben als für die gewünschte Operation nötig wäre. Solche zusätzliche Arbeit benötigt Zeit, wodurch die benötigte Zeit verlängert wird, um die Ausführungs-/Simulationsstufe zu erreichen.

[0009] In den wenigen Fällen aus der Versuchsliteratur, bei denen die Kompilierungsbetriebsart die gesamte Hilfsstufe im Speicher behält, eher als in einer Sequenz von Zwischendateien, ist es in dem Speicher eines einzelnen Prozessors gewesen. Wohingegen sich ein globaler Zugriff auf die gesamte Zwischenstufe durch den Kompilierungsarbeitsgang hindurch als wesentliche Ausführungs-/Simulationsleistungssteigerung erwiesen hat, besitzt irgendein einzelner Prozessor allgemein einen begrenzten Bereich von adressierbarem als auch physikalisch vorhandenem Speicher. Somit begrenzen solche Ansätze die Leichtigkeit, mit der neue Agenten eingefügt werden können, um den Kompilierarbeitsgang zu verändern oder eine neue Simulations-/Ausführungsvorrichtung anzustreben, und die Größe des Programms oder Modells, das auf einem Einzelprozessor kompiliert werden kann.

[0010] Innerhalb der bestehenden Kompilerliteratur und der bestehenden Herstellungskompilerumgebung wird die Kompilierung entweder parallel ausgeführt, indem Mehrfachprozessoren mit gemeinsamem Speicher verwendet werden, um eine einzelne Phase der Kompilierung zu beschleunigen, oder es werden Quelldateien unabhängig in ein zugehöriges Ausführbares kompiliert, gefolgt von einer sequenziellen Verknüpfung der Binärdaten in ein einziges Ausführbares. Die Kompilierung mittels der Beschleunigung einer einzelnen Kompilierungsphase auf einem Mehrfachprozessor mit gemeinsamem Speicher ist für Forschungszwecke gut geeignet, ist jedoch nicht direkt anwendbar zur Verringerung der gesamten Kompilierungsverzögerung oder einer inkrementellen Rekompilierungsverzögerung. Die Kompilierung von jeder Datei isoliert, die ein Mehrdateienprogramm oder Modell umfaßt, gestattet keinen Fluß von Informationen zwischen Dateien, um ein optimaleres Ausführbares zu erhalten. Zum Beispiel ist der in einer Datei vorliegende Körper einer Funktion nicht für die Einarbeitung an dem Aufrufort in einer anderen Datei (oft als in-lining bekannt) verfügbar, wenn nicht der Körper textlich als Teil der Kompilierung der zweiten Datei umfaßt ist. Wenn mehr Informationen textlich in einer einzelnen Datei umfaßt sind, steigert die Dateigröße, welche schließlich die gesamte Programm- oder Modellgröße begrenzt, die kompiliert werden kann, die gesamte Menge an Arbeit, die für das Kompilieren benötigt wird (da die gleiche Information mehr als einmal während der Kompilierung analysiert wird).

[0011] 1990 wurde Forschung veröffentlicht, die die Darstellung eines analysierten Hardwarebeschreibungssprachenmodells beschreibt, welches Hilfsdarstellungsinstanzen abstrakter Datentypen (Klassen) verwendet. Speicheradressen (Zeiger) beschreiben die Beziehung zwischen den Instanzen. Zum Beispiel kann eine Sequenz von Hilfsdarstellungsinstanzen jeweils einen Zeiger auf den nächsten aufweisen, wodurch eine verknüpfte Liste gebildet wird. Diese Arbeit behandelte nicht das Aufteilen einer Hilfsdarstellung über mehr als einen Knoten (virtuellen Adressraum), noch integrierte sie mehr als die Darstellung der Analysenphase des Kompilers.

[0012] 1991 wurde weitere Forschung veröffentlicht, die die Machbarkeit des Kompilierens, Simulierens und Ausführens von Hardwaremodellen untersuchte, die gemeinsamen Speicher oder mitteilungsbasierte Parallelprozessoren mit einer parallelen Hilfsdarstellung verwenden. Diese Veröffentlichung schlug die Verteilung einer Hilfskompilerdarstellung vor, indem jeder Zeiger in der Hilfsdarstellung der analysierten Form durch ein Tuple (Satz) ersetzt wird, das aus einem Feld, welches den Knoten bezeichnet, und einem Feld besteht, welches die Hilfsdarstellungsadresse des bestimmten Knotens bezeichnet. Diese Arbeit untersuchte ebenso die Komplexitäten und möglichen Ansätze für eine inkrementelle Kompilierung.

[0013] Eine Veröffentlichung von 1993 berichtet

ohne ein weiteres Implementierungsdetail über eine Fortentwicklung der Arbeit von 1991 bei der Beschreibung einer verteilten Nachanalysehilfsdarstellung und eine sorgfältige Nachausarbeitung und Nachoptimierung- (in-lining) Neuverteilung der Prozesse innerhalb der Hilfskompilierung. Diese Arbeit erörterte nicht eine einzelne, kompilerorientierte Datenbank, die mehrere Kompilierungsphasen, Simulation und Ausführung umspannt, und erörterte nicht die Paralleldatenbank-Darstellung.

[0014] Weitere Arbeit offenbarte eine verteilte, kompilerorientierte Datenbank mit Klienten, die umfaßt:

– Quellenanalysatoren (Kompiler-Bestandteil)
– Nachbearbeiter (Kompiler-Bestandteil)
– Optimierer (Kompiler-Bestandteil)
– Codeerzeuger (Kompiler-Bestandteil)
– Assemblierer (Kompiler-Bestandteil)
– Verknüpfer (Linker) (Kompiler-Bestandteil)
– Laufzeitsystem (Simulations-/Ausführungs-Bestandteil)
– Tester (Debugger) (Simulations-/Ausführungs-Bestandteil)
– Profilkorrektureinrichtungen (Simulations-/Ausführungs-Bestandteil)
– Ereignisprotokoll (Simulations-/Ausführungs-Bestandteil)
– Grafische Werkzeuge (Bestandteile von verschiedenen Phasen)

[0015] Die Arbeit führte das Konzept einer einzelnen, kompilerorientierten Datenbank, die die Kompilierung und Simulation/Ausführung auf einem Computer mit mehreren Knoten umspannt, ein.

[0016] Zusammengefaßt ist eine Vorrichtung mit Kompilier- und Simulations-/Ausführungsbetriebsart wünschenswert, die effizient einen globalen Zugriff auf bestimmte Informationen, die sowohl für die Kompilierung als auch die Simulation/Ausführung benötigt werden, für die Prozessoren, den Speicher und die optional rekonfigurierbare Logik von einem oder mehreren Verarbeitungsknoten liefert, sowie solche Knoten für die Nutzung verfügbar werden. Solch eine Vorrichtung und solche Arbeitsbetriebsarten würden eine Kompilierung und Simulation/Ausführung größerer Designs, als sie mit Hilfe einer Kompilierung auf einem einzelnen Knoten bewerkstelligt werden kann, zur Verfügung stellen, während die Möglichkeiten für eine globale Optimierung und eine inkrementelle Neukompilierung geschaffen werden, was die Zeit verringert, die benötigt wird, um sowohl zu kompilieren als auch zu simulieren/auszuführen.

Kurze Beschreibung der Erfindung

[0017] Diese Erfindung, wie sie in den angehängten Ansprüchen definiert ist, offenbart eine kompilerorientierte Datenbank und eine Klientenvorrichtung, die effiziente und eng integrierte Betriebsarten schaffen, einschließlich einer Kompilierung, Simulation und Ausführung von Computerprogrammen und Hardwaremodellen. Die Erfindung nutzt eine oder mehrere Knoten eines Parallelcomputersystems, wobei jeder Knoten ein einzelner Prozessor oder ein Mehrfachprozessor mit gemeinsamem Speicher ist, der optional um rekonfigurierbare Logikvorrichtungen vergrößert ist. Vorteile der Erfindung umfassen die Fähigkeit, Designs zu kompilieren, die vielmals größer sind, als sie ein Knoten aufnehmen könnte, während ein globaler Zugriff auf Hilfsinformationen geschaffen wird, die mit dem Kompilierungs- oder Simulations-/Ausführungsbetrieb verknüpft sind. Dieser globale Zugriff verringert die benötigte Zeit signifikant, um ein Computerprogramm und/oder Hardware-Modell zu (re-) kompilieren und zu simulieren/auszuführen.

Kurze Zusammenfassung der Zeichnungen

[0018] Die Erfindung wird im folgenden anhand von Ausführungsbeispielen unter Bezugnahme auf eine Zeichnung näher erläutert. Hierbei zeigen:

[0019] Fig. 1 ein Beispiel der zugrundeliegenden Hardwarevorrichtung, einschließlich zwei Mehrfachprozessorknoten mit einem gemeinsamen Speicher mit optional rekonfigurierbaren Logikblöcken;

[0020] Fig. 2 ein Beispiel eines optionalen, unabhängigen rekonfigurierbaren Logikblocks innerhalb der zugrundeliegenden Hardwarevorrichtung;

[0021] Fig. 3 ein Beispiel eines optionalen rekonfigurierbaren Logikblocks, der als ein Teil eines oder mehrerer Prozessoren verbunden ist;

[0022] Fig. 4 die Verbindung von Beispielklienten mit der verteilten, kompilerorientierten Datenbankvorrichtung;

[0023] Fig. 5 Mittel, mit welchen ein Datenbankobjekt eine Referenz auf ein zweites, lokal aufgelöstes Datenbankobjekt herstellt und ein drittes Proxy-Datenbankobjekt eine Referenz auf ein viertes Objekt herstellt, das nur auf einem entfernten Knoten innerhalb der kompilerorientierten Datenbankvorrichtung vorliegt;

[0024] Fig. 6 Mittel, mit denen ein Proxy-Datenbankobjekt eine lokale Kopie eines entfernten Objekts innerhalb der kompilerorientierten Datenbankvorrichtung in einen Cache-Speicher kopiert;

[0025] Fig. 7 Mittel zum Liefern einer Kohärenz zwischen mehreren entfernten Proxies innerhalb der kompilerorientierten Datenbankvorrichtung;

[0026] Fig. 8 Mittel zum Lokalisieren einer ersten nicht lokalen Kopie eines bestimmten Objekts inner-

halb der kompilerorientierten Datenbankvorrichtung;

[0027] Fig. 9 Mittel zum Lokalisieren einer lokalen Darstellung kanonisch definierter Objekte innerhalb der kompilerorientierten Datenbankvorrichtung;

[0028] Fig. 10 Mittel, mit denen die Klientenfunktionalität in zwei oder mehr Betriebssystemprozesse geteilt werden kann.

Beschreibung der bevorzugten Ausführungsform

[0029] Dieser Abschnitt beschreibt die beste Ausführungsform der Erfindung, die den Erfindern derzeit bekannt ist, jedoch werden Fachleute erkennen, daß es viele verwandte Implementierungen gibt, die die gleiche Erfindung umsetzen, vielleicht indem alternative Mittel zum Erreichen der Funktionalität eines Bestandteils ersetzt werden.

[0030] Die zugrundeliegende Hardwarevorrichtung wird in den Fig. 1, Fig. 2 und Fig. 3 vorgestellt. Fig. 1 zeigt die gesamte Vorrichtungsstruktur. Figur zeigt ein Detail eines optionalen Blocks **9**. Fig. 3 zeigt eine optionale rekonfigurierbare Logik in den Blöcken **1** und **2**.

[0031] In Fig. 1 repräsentieren die Blöcke **1** bis **8** einen herkömmlichen, cache-kohärenten Mehrfachprozessor. Bei der Auslassung der Blöcke **2**, **4** und **6** kann die Hardwarevorrichtung ebenso die Form eines Einprozessorknotens annehmen, der die Blöcke **1**, **3**, **5**, **7** und **8** enthält. Diesem herkömmlichen Einprozessor kann eine Nachrichtenschnittstelle (Block **10**) zugefügt werden, um eine herkömmliche massiv parallele Maschine zu implementieren, die Protokolle, wie den IEEE-Standard 1596 Scalable Coherent Interface, benutzt. Herkömmliche Eingabe-/Ausgabevorrichtungen, zum Beispiel eine grafische Nutzerschnittstelle, ein lokales Netzwerk und ein Diskuntersystem, werden als vorliegend angenommen (obwohl sie nicht gezeigt sind), die herkömmliche Mittel verwenden, die Fachleuten wohl bekannt sind.

[0032] Rekonfigurierbare Logikblöcke, Block **22**, können optional in den herkömmlichen Einprozessor, den Mehrfachprozessor mit gemeinsamem Speicher oder einen Massivparallelprozessor eingefügt werden, wie oben beschrieben ist. Als isolierte Vorrichtungen sind rekonfigurierbare Logikblöcke in einer großen Vielzahl von Variationen erhältlich, für die sowohl das Verhalten einzelner Zellen und Zwischenverbindungen verändert werden kann (rekonfiguriert), während die Vorrichtung in einem System installiert ist, manchmal auf einer Zyklus-für-Zyklus-Basis.

[0033] Um die Kompilierung und Simulation/Ausführung durch eine kompilerorientierte Datenbank zu unterstützen, können ein oder mehrere rekonfigurierbare Logikblöcke optional in ein System integriert werden, indem eine Schnittstelle für gemeinsamen Speicher (wie in Block **9**), eine nachrichtenbasierte Schnittstelle (wie in Block **9**) oder als ein Teil der Ausführungspipeline verwendet wird, die für einen oder mehrere Prozessoren (Blöcke **30** und **22** in Fig. 3) verfügbar ist. Ohne einen Verlust an Allgemeinheit können ein oder mehrere rekonfigurierbare Logikblöcke (Block **21**) durch herkömmliche Speicheranordnungen (memory arrays), festgelegte Logikvorrichtungen (zum Beispiel einen Analog-nach-digital-Konverter), die dem Kompiler und/oder den Steuereinrichtungen bekannt sind, ersetzt werden.

[0034] Block **9** stellt die interne Struktur von einem (oder mehreren) rekonfigurierbaren Logikvorrichtungen (Block **21**) ausführlich dar, die in die Hardwarevorrichtung über einen gemeinsamen Speicher oder eine Nachrichtenweiterleitungsschnittstelle zu der lokalen Zwischenverbindung (Block **20** bis Block **7**) integriert sind. Um das Verhalten und die Zwischenverbindung von Block **9** und Block **21** zu rekonfigurieren, können eine oder mehrere Adressen in die Zwischenverbindungs- (Block **7**) Speicherabbildung oder Eingabe-/Ausgabeabbildung abgebildet werden, so daß die Rekonfigurationsinformation von Block **9** geschrieben (oder gelesen) werden kann. Ein Mittel des Schreibens (und Lesens) solch einer Zwischenverbindungsinformation ist es, der Konfiguration von jedem rekonfigurierbaren Logikgatter/element und Zwischenverbindungspunkt innerhalb des Blocks **21** eine bestimmte Speicheradresse zuzuweisen. Der geschriebene (oder gelesene) Wert aus der Adresse bestimmt das Gatter-/Element-Verhalten oder die Zwischenverbindungswegsuche (routing). Ein alternatives Mittel ist es, zwei Adressen zu verbinden, wobei eine benutzt wird, um ein bestimmtes Gatter/Element oder einen Zwischenverbindungsweiterleitungspunkt zu spezifizieren, und wobei die andere Adresse benutzt wird, um einen Wert des Gatters/Elements oder des Zwischenverbindungsweiterleitungspunkts zu spezifizieren. Alternativ kann die gleiche Konfigurationsinformation mittels eines Prozessors über die Nachrichtenzwischenverbindung (Block **13**) und die Nachrichtenschnittstelle (Block **10**) geschrieben (gelesen) werden. Andere Mittel des Rekonfigurierens des Logikblockes **9** sind Fachleuten aus einer ausgedehnten Geschichte des Abbildens von Eingabe-/Ausgabe- und speicherabgebildeten Eingabe-/Ausgabevorrichtungen in einen Mehrfachprozessor mit gemeinsamem Speicher oder Architekturen von Massivparallelprozessorsystemen bekannt.

[0035] Sobald die rekonfigurierbare Logik (Block **9**) mit Hilfe von Software konfiguriert worden ist, die auf einem oder mehreren Prozessoren abläuft, kann die Schnittstelle Block **9** verwendet werden, so daß die rekonfigurierbare Logik (Block **9**) in der Lage ist, das Verhalten von (eingebetteten) Prozessen auszuwer-

ten, indem die gleichen Formen der Zwischenprozeß- und Zwischenprozessorkommunikation verwendet werden, die gewöhnlich von Prozessoren mit gemeinsamem Speicher und Massivparallelprozessoren verwendet werden. Diese Mechanismen, die dem Fachmann wohl bekannt sind, umfassen Punkt-zu-Punkt- und Mehrfachaussendungs-Nachrichten, Interrupts, kohärente Zugriffe auf gemeinsamen Speicher. Im Ergebnis kann der Block **9**, sobald er einmal rekonfiguriert ist, in der Systemarchitektur als ein hoch paralleler Mehraufgabenprozessor mit einem festgelegten Programm teilnehmen.

[0036] Alternativ können die rekonfigurierbaren Logikblöcke, die oben beschrieben wurden, (Block **9**) in die Anweisungsausführungspipeline von einem oder mehreren Prozessoren (Block **1**) integriert werden, wie in Fig. 3 gezeigt ist. Eine oder mehrere Anweisungssatzcodierungen (allgemein als "OP-Codes" bekannt) können für Funktionalität reserviert werden, die mit Hilfe eines optionalen Koprozessors, einer Zwischenverbindung, einem Arbiter und einem rekonfigurierbaren Logikblock (Blöcke **9** und **30**) implementiert wird. Wenn solch ein OP-Code während der Ausführung eines Anweisungsstroms innerhalb eines Prozessors erreicht wird, werden der OP-Code, erweiterte OP-Codes (falls es welche gibt) und Eingabeoperanden von dem Prozessor (Block **1**) abgerufen, über die Koprozessorschnittstelle, die Zwischenverbindung und den Arbiter an einen geeignet konfigurierten rekonfigurierbaren Logikblock (Block **9**) gesandt, wobei der rekonfigurierbare Logikblock die mit Hilfe der von dem Prozessor gelieferten Informationen bezeichnete Operation ausführt und zu einem späteren Punkt einen Vervollständigungsstatus und optionale Ergebnisse an den Prozessor zurückgibt, der die Operation (Block **1**) verursacht hat, wodurch diesem Prozessor gestattet wird, die Ausführung der Originalanweisung (möglicherweise außerhalb der Reihenfolge) zu beenden. Details des Arbiters und der Schnittstelle sind Fachleuten aus den Designs, beispielsweise denen der Koprozessor-Schnittstelle der 68.000-Familie oder von enger integrierten proprietären Schnittstellen, gut bekannt.

[0037] Anweisungen, die von dem rekonfigurierbaren Logikblock ausgeführt werden, können verwendet werden, um daß das Gatter-/Elementverhalten oder die Zwischenverbindungswegsuche innerhalb des rekonfigurierbaren Logikblocks, des Zugriffsspeichers, der Steuereinrichtung oder Vorrichtungen, die innerhalb des rekonfigurierbaren Logikblocks eingebettet sind, oder einen Lesezustand zu rekonfigurieren, der aus früheren Operationen innerhalb des rekonfigurierbaren Logikblocks resultiert. Wie bei dem gemeinsamen Speicher und der nachrichtenbasierten Verwendung des Blocks **9** sind die Klienten der kompilerorientierten Datenbank für ein Wiedererkennen installierter rekonfigurierbarer Logikblöcke (einschließlich ihrer Fähigkeiten über einen fest verdrahteten Konfigurationszustand) und ein Erzeugen einer geeigneten Rekonfigurationsinformation verantwortlich, wie es herkömmlich getan wird, wenn ein Kompiler Anweisungen in eine Datei (oder einen Speicher) für einen herkömmlichen Prozessor ausgibt.

[0038] Der Speicher (Block **8**), die Cache-Speicher (Block **3** und **5**) und schließlich der Prozessor (Block **1**) der darunterliegenden Hardwarevorrichtung können dann programmiert werden, um eine verteilte, kompilerorientierte Datenbank und eine zugehörige Klientvorrichtung zu implementieren, wie sie in Fig. 4 gezeigt ist. Die sich ergebende Vorrichtung kann dann bei der Kompilierung als auch bei einer Simulations-/Ausführungsbetriebsart verwendet werden, um größere Programme/Hardwaremodelle in weniger Zeit zu kompilieren und zu simulieren/auszuführen.

[0039] Die verteilte, kompilerorientierte Datenbank behält eine Sammlung von Hilfsdarstellungsdatenbankobjekten, die besonders für die Kompilierung und/Simulations-/Ausführungsbetriebsart der Erfindung geeignet sind. Erläuternde Beispiele der Datenbankobjekte umfassen ein Objekt, das einen Buchstabenidentifizierer "i" bezeichnet, eine Erklärung, einen Ausdruck, eine gleichzeitige Anweisung, einen ausgearbeiteten Prozeß, einen ausgearbeiteten Treiber, den gegenwärtigen Zustand eines Prozesses, der simuliert/ausgeführt wird, die Zeit/Wertänderungsgeschichte einer Zwischenprozeß-Nachrichten-Schlange oder einer gemeinsamen Variablen über irgendein Zeitintervall oder den Inhalt einer nutzerdefinierten Datei, die zum Lesen oder zum Schreiben von einem ausgeführten Programm und/oder Modell geöffnet wurde. Die Datenbank kann über den Speicher, Auslagerungsraum und das Dateisystem vieler Verarbeitungsknoten verteilt sein. Jedes Objekt hat einen einzigartigen wahrnehmbaren Wert, der über die verteilte, kompilerorientierte Datenbank kohärent ist, ungeachtet dessen, welcher Rechnerknoten die Referenz ausführt.

[0040] Allgemeine Techniken zum Implementieren der verteilten, kompilerorientierten Datenbank sind dem Fachmann durch die Anpassung von Allzweckparalleldatenbanken oder aus der Literatur über verteilten, gemeinsamen Speicher gut bekannt. Einzigartige Aspekte dieser Erfindung umfassen eine Vorrichtung für die Kompilier- und Simulations-/Ausführungsbetriebsarten (z. B. Klienten, die beispielhaft durch die Blöcke **40** bis **50** dargestellt sind), die in Allzweckdatenbanksystemen nicht vorhanden sind, und Vorrichtungen zum Kopieren der bestimmten Kompiler- oder Simulations-/Ausführungsobjekte in den Cache-Speicher, die von den Kompilierungs- und Simulations-/Ausführungsbetriebsarten benötigt werden. Im Gegensatz hierzu liefern verteilte Systeme mit gemeinsamem Speicher entfernten Zugriff auf zusammenhängende, fest Längen und feste Körnigkeitsintervalle aufweisende Ardeßräume, eher als eine

Sammlung von semantisch bedeutungsvollen, jedoch unzusammenhängenden Objekten, z. B. einen Baum von Deklarationen innerhalb eines deklarativen Bereichs der Kompilierzeitdarstellung.

[0041] Die verteilte Kompilerdatenbank-Schnittstelle liefert eine Programmiersprachenschnittstelle (Block **51**), durch den Klienten (z. B. jene, die in den Blöcken **40** bis **50** dargestellt sind) Sammlungen von kompiler- und simulations-/ausführungsbezogenen Datenbankobjekten erzeugen, Kopien von vorexistierenden Objekten anfordern, automatisch aktualisieren, herausgeben und freigeben können. Allgemeine Mittel zum Implementieren solcher Programmiersprachenschnittstellen sind Fachleuten bereits aus vorher veröffentlichter Arbeit bekannt, z. B. der AIRE/CE-Programmiersprachenschnittstelle für Einprozessorsysteme, die die VHDL-Hardware-Beschreibungssprache implementiert.

[0042] Um einige der Kompiler- und Simulations-/Ausführungsklienten zu würdigen, die für eine Verbindung mit der kompilerorientierten Datenbank geeignet sind, beschreiben die folgenden Absätzen Klienten, die in den Blöcken **40** bis **52** innerhalb des dargestellten Verarbeitungsknotens (Block **52**) dargestellt sind. Man beachte jedoch, daß nicht alle der dargestellten Klienten auf einem gegebenen Verarbeitungsknoten vorliegen müssen, noch sind andere Klienten, die mit der Kompilierung oder Simulation-/Ausführung verknüpft sind, von dem Verbinden mit der kompilerorientierten Datenbank auf einem oder mehreren Knoten ausgeschlossen.

[0043] Insbesondere ist es wahrscheinlich, daß einige Knoten Nutzer-Schnittstellen-Klienten aufweisen, während andere Knoten wahrscheinlicher Hintergrundverarbeitungs-Klienten besitzen, z. B. einen Ausarbeiter, einen Optimierer oder einen Codeerzeuger.

[0044] Der Quellcodeanalysierer (Block **40**) empfängt einen Befehl von der verteilten, kompilerorientierten Datenbankschnittstelle (Block **51**), wobei die Übersetzung eines Bruchstücks (mit dem Kontext, der mit der kompilerorientierten Datenbank verknüpft ist) oder einer kompletten Quellcodedatei, die ein Programm oder Hardwaremodell bildet, in die kompilerorientierte Datenbankdarstellung gelenkt wird. Abhängigkeiten, denen während der Analyse einer gegebenen Datei begegnet wird, z. B. einer VHDL-Nutzungsklausel, die sich auf eine andere deklarative Region bezieht, erfordern eine Verzögerung bei der Analyse, bis die kompilerorientierte Datenbank eine lokale Kopie der verwendeten Deklarationen und verknüpften Datenbankobjekte liefert. Um die nachfolgenden Verzögerungen zu minimieren, kann eine deklarative Region, die von dem Quellenanalysator auf einem Rechnerknoten verwendet wird, eine proaktive Übertragung der referenzierten deklarativen Region zu anderen Knoten anstoßen, die Quellen erzeugen. Solche Cacheabspeicherungsheuristiken sind sprachenspezifisch und nutzerspezifisch mit Hilfe von während des Betriebs erlernter Informationen gesteuert und werden mit der dauerhaften Form der verteilten, kompilerorientierten Datenbank gespeichert.

[0045] Der Ausarbeiter (Block **41**) schreibt vorher analysierte Informationen von der verteilten, kompilerorientierten Datenbank neu, um das Design und die Unterprogramm-/Funktionshierarchie abzuflachen oder um Beschränkungen zwischen einem Unterprogramm/einer Funktionsdefinition und seinen Aufruforten innerhalb anderer Teile des Programms oder des Hardwaremodells zu verbreiten. Der Ausarbeiter nutzt die globale Sichtbarkeit durch die Datenbank hindurch und die Fähigkeit Sammlungen von Datenbankobjekten erneut zu benutzen, die vor und nach der Analyse dieselben bleiben. Z. B. kann der Analysator den Satz von Deklarationen in einem Prozeß, die einen konstanten Typ und eine konstante Größe haben, von jenen mit variierendem Typ oder variierender Größe abhängig von dem Punkt trennen, an dem eine Instanz erzeugt wird oder ein Unterprogramm/eine Funktion aufgerufen wird. Der Ausarbeiter erzeugt dann nur neue, ausgearbeitete Datenbankobjekte für jene Objekte mit dem spezifischen Typ oder der spezifischen Größe, indem die verteilte, kompilerorientierte Datenbank verwendet wird, um den Speicher- und Objekterzeugungs-/Wartungsüberbau zu verringern. Der Ausarbeitungsklient kann auch die verteilte Datenbank ausnutzen, um unter Ausarbeitungen von Komponenteninstanzen Objektinstanzen oder Unterprogrammhierarchien auf anderen Knoten zu erzeugen. Indem Komplexitätsmetriken verwendet werden, die während der Analyse geschätzt wurden und mit der Datenbank gespeichert sind und rekursiv den Ausarbeitungsbaum hinauf verbreitet werden, kann der Ausarbeiter die Heuristiken verwenden, um die Ausarbeitung auf andere Verarbeitungsknoten zu erstrecken, so daß das Ergebnis der Ausarbeitung ein guter erster Durchlaufteil des Programms oder Designs auf anderen Prozessoren ist.

[0046] Der Optimierer-Klient (Block **42**) nutzt die globale Sichtbarkeit in die Designdatenbank, um eine verteilte, globale Analyse und Neuschreibungstransformationen zu ermöglichen, wie jene, die von dem Erfinder in den vorherigen Veröffentlichungen (J. Willis, "Optimizing VHDL Compilation for Parallel Simulation", Carnegie Mellor University Dissertation, Pittsburgh, Pennsylvania, 1991; J. Willis et al., "Optimizing VHDL Compilation for Parallel Simulation", IEEE Design & Test of Computers, Sep. 1992, Seiten 42–53) beschreiben wurden. Ohne die selektive globale Sichtbarkeit in die gesamte analysierte, ausgearbeitete und Simulations-/Ausführungsdatenbank müssen Optimierer Entscheidungen mit weniger voll-

ständiger Information fällen, was allgemein zu weniger optimalen Optimierungen führt. Eine direkte Kommunikation über die Datenbank mit anderen Optimierern gestattet das Teilen einer Hilfsoptimierungsanalyse, z. B. der Datenflußanalyse eines Unterprogramms. In Abwesenheit der offenbarten verteilten, kompilerorientierten Datenbank müssen die Optimierer, die getrennte parallele Kompilierungen ausführen, häufig Analysen von gemeinsamen Komponenten und Unterprogrammen erneut erzeugen.

[0047] Der Abbildungs-Klient (Block **43**) iteriert die Zuweisung von Arbeitslast auf Verarbeitungsknoten, die während der parallelen Ausarbeitung initiiert wird, und beraumt quasi statisch die Evaluation von mehreren Ausführungsfäden (Prozessen) auf jedem Verarbeitungsknoten an (J. Willis et al., "Min Sim: Optimized, Compiled VHDL Simulation Using Networked & Parallel Computers", Proceedings of Fall 1993 VHDL International User's Forum). Eine kritische Pfadanalyse, die in der kompilerorientierten Datenbank kommentiert ist, liefert erneut den globalen Gesichtspunkt, von dem aus ein Effekt, eine quasi statische Abbildung und Zeitplanung erreicht werden kann. Die Abbildungs-Klienten auf jedem Verarbeitungsknoten mit Simulations- oder Ausführungsarbeitslast tauschen Arbeitslast aus, bis sie bei einer Lastbalance ankommen, die vorausgesagt ist, um die Simulations-/Ausführungsrate zu maximieren. In der Abwesenheit von solch einer globalen, Kompilierungszeitinformation hängen parallel Simulations-/Ausführungsumgebungen von einer Laufzeitlastausgleichung ab, was zu einem größeren Laufzeitüberbau und einer verringerten Simulations-/Ausführungsleistung führt.

[0048] Der Codeerzeugungs-Klient (Block **44**) folgt dem Abbildungen und dem Zeitplanen, indem ein Pseudocode, Prozessorausführbares oder rekonfigurierbare Logikkonfigurationsinformation direkt in den Adreßraum des Codeerzeugers, in einen verknüpften Betriebssystemprozeß auf demselben Verarbeitungsknoten oder in eine herkömmliche Datei für eine spätere Ausführung ausgegeben werden. Das Ausgeben des Ausführbaren und der Konfigurationsinformation direkt in denselben Betriebssystemprozeß wie der Codeerzeuger oder ein eng verkoppelter Prozeß liefert eine schnelle, lokalisierte und inkrementelle Antwort auf Änderungen in der verteilten, kompilerorientierten Datenbank. Änderungen können mit Hilfe von Unterbrechungspunkten für ein symbolisches Testen, Einfügen von Profilierungscode, Einfügen von absichtlichen Fehlern (zur Fehlersimulation) oder anderen Deltas auf der Datenbank angetrieben werden, die auf die Simulation/Ausführung einwirken.

[0049] Der Verknüpfer/Lader (linker/loader) (Block **45**) arbeitet eng sowohl mit der kompilerorientierten Datenbank als auch dem Codeerzeuger zusammen, um Adressen von symbolischen Werten in tatsächliche Speicheradressen aufzulösen, um Codefragmente zu verschieben, um die Cache-Leistung zu verbessern, um Cachesteuerungsanweisungen in das Ausführbare einzufügen und um externe Büchereien (libaries) in den Adreßraum zu verknüpfen/zu laden, der ebenso Ziel von dem Codeerzeuger ist. Die Verwendung einer gemeinsamen, kompilerorientierten Datenbank liefert ein schnelles, feinkörniges Zusammenarbeiten zwischen dem Codeerzeuger und dem Zeitplaner (scheduler), welches unausführbar wäre, falls sie mittels herkömmlicher, getrennter Betriebssystemprozesse verwirklicht würden, die durch eine Datei oder eine unidirectionale Pipeline kommunizieren.

[0050] Der Ausführungs-/Simulations-Klient (Block **46**) nutzt sowohl symbolische als auch Kontrollinformationen aus der kompilerorientierten Datenbank, um ordnungsgemäß lokale Speicher, gemeinsame Speicher und rekonfigurierbare Logik zu laden und die Ausführung zu steuern. Der Ausführungs-/Simulations-Klient profitiert erneut von dem feinkörnigen Zugriff auf die globale, kompilerorientierte Datenbank und den Adreßraum, der als ein Ziel von dem Codeerzeuger und Verknüpfer/Lader verwendet wird. Programmsprachenschnittstellen, die eine Modifizierung des Programms oder Designs erfordern, das simuliert wird, können schnell und effizient durch Aufrufe aus dem Simulations-/Ausführungsadressraum aufgenommen werden, was zu Änderungen an der kompilerorientierten Datenbank und einem inkrementellen Neukompilierungsbefehl führt, der an den Codeerzeuger und Verknüpfer-/Laderklient gesandt wird. Solche engen Modifizierungszyklen sind mit herkömmlichen Kompilierungs- und Ausführungsumgebungen unausführbar, wo die Kompilier- und Ausführungs-/Simulationsumgebung nicht simultan aktiv sind und in einer bidirektionalen Kommunikation stehen; als ein Ergebnis muß der ursprünglich erzeugte Code eine erhebliche Flexibilität aufweisen und somit eine reduzierte Simulations-/Ausführungsleistungsfähigkeit, um die benötigte Modifikation zu gestatten. Steuerungsanweisungen entspringen der Nutzerschnittstelle oder dem Tester (debugger) und kommunizieren kohärent durch die kompilerorientierte Datenbank hindurch an jeden Ausführungs-/Simulations-Klienten auf jedem Verarbeitungsknoten, der an der parallelen Kompilierung und Simulation/Ausführung teilnimmt.

[0051] Der Tester-Klient (Block **47**) verbindet den Nutzer und die kompilerorientierten Datenbank, um die Ausführung, Unterbrechungspunkte, einen Setzzustand und einen Lesezustand zu steuern. Während herkömmliche Tester direkt in das Ausführbare hineinreichen, gestattet der kompilerorientierte Datenbankansatz dem Tester, geeignete Änderungen in der kompilerorientierten Datenbank von einem Verarbeitungsknoten (mit einer geeigneten Nutzerschnittstelle) auszuführen und Befehle über die Datenbank

zu initiieren, die von dem Quellcodeanalysierer durch den Codeerzeuger und Verknüpfer/Lader gesehen werden. Da relativ wenig Code tatsächlich in einem gegebenen Simulations-/Ausführungsdurchgang getestet wird, gestattet dieser Ansatz den meisten Code für eine maximale Leistungsfähigkeit zu erzeugen; wenn ein Test benötigt wird, können die geeigneten Codefragmente rekompiliert und zustandsabgebildet werden, um eine Region des Ausführbaren mit suboptimaler Leistungsfähigkeit, jedoch guter Korrelation mit dem originalen Quellcode zu erzeugen. Unterstützt von der vollen Information, die in der kompilerorientierten Datenbank enthalten ist, sind der Codeerzeuger und Verknüpfer bestens ausgestattet, um einen Zustand auf den Stapel, die Signalschlangen und die statischen Gebiete zwischen optimierte und Testcodesequenzen abzubilden. Solch ein Abbilden ist für einen Tester schwierig oder unmöglich, der direkt auf einer ausführbaren Abbildung arbeitet und der nur mit Hilfe einer herkömmlichen Symboltabelle geführt wird.

[0052] Das Testen von Simulationen bedeutet kompliziertes Optimieren und Abbilden und Planen von Transformationen, die versuchen Hardwaremodelle in einer lokalen "zyklusgetriebenen" Betriebsart auszuführen, wobei, wo möglich, Informationsabhängigkeitsinformationen, jedoch nicht lokale Zeitgebungsdetails beibehalten werden. Wenn das Testen, Profilieren oder eine andere Nutzerschnittstellenfunktion ein Gebiet eines zyklusgetriebenen Codes von einem Voll-Zeitfestlegungsabschnitt der Simulation betreten, erfordert das Erhalten sichtbarer Semantiken des Hardwaremodells eine lokale Code- und Zustandstransformation von der zyklusgetriebenen auf die Voll-Zeitfestlegungssemantik, die wiederum mit Hilfe der fertigen Verfügbarkeit der kompilerorientierten Datenbank möglich gemacht wird. Allgemein wird ein Zeitfestlegungszustand erzeugt, indem lokal die Simulation von dem bekannten Zustand in den neuen Voll-Zeitfestlegungs-Code erneut ausgeführt wird, um einen Voll-Zeitfestlegungszustand zum Testen zu erzeugen. Solch eine testorientierte, inkrementelle Zeitfestlegungsfähigkeit ist eine wichtige Quelle der Leistungsfähigkeit, wenn die Bibliotheken mit wesentlichen Zeitfestlegungsdetails kompiliert werden, von denen nur ausgewählte Abschnitte für die Hardwaremodellleistungsfähigkeit kritisch sind. Diese Situation tritt oft auf wenn VHDL's VITAL Zeitgebungsgrundelemente kompiliert und simuliert werden. Nachdem das Testen, das Profilieren und andere Nutzerschnittstellenaktivitäten die Region des in Frage stehenden Codes verlassen, kann die höhere leistungsfähigere Codeimplementierung wiederhergestellt werden, um die Simulationsleistung zu steigern. Solch eine schnelle und lokalisierte Änderung des Ausführbaren hängt von der feinkörnigen, kompilerorientierten Datenbankstruktur für sowohl die Daten- als auch für die Befehlskommunikation ab.

[0053] Der Profilier-Klient (Block **48**) liefert dem Nutzer Mittel, um das Gesamtsimulations- und Ausführungsverhalten zu untersuchen. Dies geschieht im Gegensatz zu einem Tester, der typischerweise eine kleine Anzahl von ausführbaren/Simulationsregionen untersucht, sehr detailliert. Das Profilieren kann über die Codepfadwege berichten, die tatsächlich von einem bestimmten Simulations-/Ausführungsdurchlauf durchlaufen wurden, über die Zeit, die in jeder ausführbaren/Simulationsregion verbracht wurde oder sogar die Gesamteigenschaften von Werten berücksichtigen, die einem bestimmten Zustand zugewiesen sind. Besonders die Fähigkeit, falls komplexe Profilierungskriterien festgestellt werden, schnelle Änderungen an dem Ausführbaren/der Simulation vornehmen zu können, die durch die kompilerorientierte Datenbank ermöglicht werden, ist erneut kritisch wichtig.

[0054] Der Nutzer-Schnittstellen-Klient, Block **49**, liefert eine Nutzersteuerung über den gesamten Kompilierungs- und Ausführungs-/Simulationsprozeß. Erneut kann mittels des Verbindens mit anderen Klienten über die kompilerorientierte Datenbank eine breite Vielzahl von Nutzerschnittstellen ohne eine direkte Abhängigkeit von anderen Klientenschnittstellen entwickelt werden. Dies ermöglicht eine unabhängige Entwicklung von Kompiler-Steuer-Schnittstellen, Kommandointerpretierern, schematischen Anzeigeschemata, Wellenformanzeigen und Analysewerkzeugen. Da die Informations- und Befehlsprotokolle von der kompilerorientierten Datenbankbuchse und nicht den anderen besonderen Klienten definiert werden, die installiert sind, hat sich die Nutzer-Schnittstellen-Entwicklung stark vereinfacht, wodurch entweder die Entwicklungskosten der Nutzer-Schnittstelle verringert werden oder eine gesteigerte Funktionalität ermöglicht wird.

[0055] Der Datei-Schnittstellen-Klient, Block **50**, sorgt für die Verteilung der Dateieingabe/-ausgabe über die Nachrichtenzwischenverbindung und den kohärenten Wiederzusammenbau der Dateieingabe/-ausgabe, die aus der Ausführung/Simulation resultiert. Der Dateizugriff über die kompilerorientierte Datenbank erleichtert die Erzeugung von optimalen Eingabe-/Ausgabe-Routinen für nutzerdefinierte Typen relativ zu einer Laufzeitzusammenstellung von Zusammenstellungstyp-Eingaben/-Ausgaben, vermeidet die Notwendigkeit für die Dateischnittstelle, sich direkt mit zwischengemischtem Lesen und Schreiben zu befassen, das auf der Hardware mit getrennten und nicht kompatiblen Datentyp-Codierungen ausgeführt wird. Der kritische Ermöglichungsschritt ist, daß die Kommunikationsschnittstelle ebenfalls semantische Information über die Information, die übertragen wird, besitzt, was in herkömmlichen Umgebungen fehlt, die die Kompilierung und Ausführung/Simulation trennen.

[0056] Die obigen Klienten stellen eine einzigartige und neue Vorrichtung und die Betriebsarten dar, die ermöglicht werden, indem Klienten durch eine kompilerorientierte Datenbank, eher als durch herkömmliche grobe unidirektionale Datei- (oder pipeline-) Verknüpfungen verknüpft werden. Einige Fachleute des Parallel-Datenbank-Designs können bei der Konstruktion solch einer kompilerorientierten Datenbank Anleihen, im Rahmen einer Erweiterung, wesentlicher Technologien machen. Um jedoch die Erörterung konkret zu machen, diskutiert der folgende Abschnitt Mittel zum Implementieren einer verteilten, kompilerorientierten Datenbank.

[0057] Fig. 5 bis Fig. 9 stellen ein Mittel zum Implementieren einer verteilten, kompilerorientierten Datenbank dar. Einer der kritischsten Punkte bei der Implementierung solch einer Datenbank ist das Vorsehen eines effektiven Mittels, Objekte zu referenzieren, die ursprünglich erzeugt wurden oder gleichzeitig auf einem anderen Knoten "ansässig" sind. Bisherige Ansätze umfaßten ein Bezeichnen jeden Zeigers mittels eines <Knoten, Adresse>-Tuples direkt oder innerhalb einer einzelnen Speicheradresse. Ein Verwenden des ersten Ansatzes verzögert jeden Zeigerzugriff in der Datenbank, um zu bestimmen, ob der Knotenbezeichner in dem Tuple mit dem gegenwärtigen Knoten korrespondiert oder ob ein entfernter Knoten benötigt wird. Der erste Ansatz verdoppelt effektiv den benötigten Speicher, um die Datenbank zu repräsentieren, da jeder Zeiger die doppelte Größe einer Speicheradresse erlangt und das meiste einer Datenbank aus Zeigern besteht. Der zweite Ansatz eliminiert die Platzbestrafung, grenzt jedoch die gesamte Datenbankgröße auf den Adressbereich ein, der mit Hilfe eines Zeigers adressiert wird. Er benötigt ferner einen sehr komplexen, maschinenabhängigen Fallen-Handhaber, der ein Rückkorrigieren einer fehlerhaften Adresse gestattet, sobald die benötigten Objekte lokal in einen Cache-Speicher gespeichert sind. Da der Fallen-Handhaber oft ein kritischer Abschnitt des Codes ist, in dem man nicht blockieren kann, während man auf eine nicht lokale Antwort wartet, sind Rückrufe ein praktisches Erfordernis, so daß die Korrektur auftritt, nachdem das nicht lokale Objekt eintrifft. Mit den obigen Nachteilen in Gedanken, wird ein dritter Ansatz in der bevorzugten Ausführungsform der gegenwärtigen Erfindung offenbart.

[0058] Jedes Objekt in der Datenbank wird mit einem bestimmten Hilfsdarstellungstyp (wie in Block **63**) verknüpft. Verschiedene Mittel zum Verknüpfen sind möglich, einschließlich einer Implementierung interner Zeiger auf die Klassendefinition (zum Beispiel eine C++ "vptr"), eine explizite ganze Zahl, die den Typ bezeichnet, oder ein Aufzählungswert. Mittels Referenz auf den Hilfsdarstellungs-Objekttyp wird der Satz gültiger Operatoren und interner Daten definiert. Beispiele für geläufige Hilfsdarstellungs-Typen umfassen Buchstaben, Deklarationen, Sammlungen, Namen, Ausdrücke, Anweisungen und eine Blockstruktur (J. Willis, "Auriga: A Compiler that Adresses NUMA Architectures", WESCON/96 IC EXPO Applications Conference on Communications and Computer Technologies, Annaheim, CA, 1996). Zusätzliche Hilfsdarstellungs-Typen, die für die verteilte, kompilerorientierte Datenbank über solche hinaus benötigt werden, die gewöhnlich intern für einen herkömmlichen Kompiler verfügbar sind, umfassen verschiedene Entferntes-Objekt-Proxies, Zwischen-Klient-Befehle, Zwischen-Klienten-Antworten und einen Simulations-/Ausführungszustand (einschließlich der Darstellung von Stapelrahmen (stack frames), statischen Gebieten und Kommunikationen innerhalb der Simulation/Ausführung).

[0059] Die meisten Zeiger beziehen sich auf ein Objekt, das auf einem Knoten lokal vorhanden ist (wie in Block **64**, eine Speicheradresse, die auf ein lokales Objekt zeigt, das mittels des Blocks **61** bezeichnet ist). Allgemein bezieht sich eine kleine Anzahl der Zeiger auf Objekte auf anderen Knoten (wie bei der Speicheradresse in Block **65**, die sich auf das entfernte Objekt des Typs 1 bezieht, das mittels Block **62** bezeichnet ist) im Wege eines entfernten (oder Proxy-) Objekts. Die erste Art eines entfernten Objekts umfaßt den tatsächlichen Typ des entfernten Objekts, das bezeichnet ist (Block **70**), den Knoten, auf dem die Referenzkopie existiert (Block **71**) und die Speicheradresse auf dem entfernten Knoten (Block **72**). Somit belegt die Mehrheit der Zeiger den minimalen Speicher, der für eine einzelne Speicheradresse benötigt wird. Eine kleine Anzahl von Außerhalb-des-Knotens-Referenzen belegen Speicher, der ungefähr die doppelte Größe (wie in Block **62** gezeigt) des <Knoten, Adresse>-Ansatzes aufweist.

[0060] In Fig. 6 ändert die verteilte, kompilerorientierte Datenbank, wenn eine Referenz (oder ein Verfahrenaufruf) auf ein entferntes Objekt des Typs 1 (Block **69**) ausgeführt wird, das für einen tatsächlichen Objekttyp (Block **70**) geeignet ist, das entfernte (oder Proxy-) Objekt in eine Typ 2-Referenz, in der der Objekttyp (Block **70** durch einen Zeiger auf eine lokale „in den Cache-Speicher kopierte" Kopie des Objekts (Block **87**) ersetzt ist. Mit einer Speicherumleitung, funktioniert die in den Cache-Speicher kopierte Kopie lokal, als ob sie auf dem lokalen Knoten resident wäre. Basierend auf dem Typ des referenzierten Objekts, dem verfügbaren Speicher, der Bandbreite als auch dem Verwendungsmuster kann die verteilte, kompilerorientierte Datenbank Domainwissen verwenden, um eine lokal in den Cache-Speicher gespeicherte Kopie von mehr als nur dem unmittelbar angeforderten Objekt abzurufen. Zum Beispiel kann die Referenz mittels Auswahl aus einer deklarativen Region die Übertragung des gesamten Inhalts der deklarativen Region, eher als nur einer Deklaration auslösen. In einem zweiten Beispiel kann das Abrufen einer Deklaration das Abrufen des Deklarati-

onstyps (und rekursiv seiner Typdefinition) auslösen. Auf ähnliche Weise kann die verteilte, kompilerorientierte Datenbank auswählen, in den Cache-Speicher kopierte Kopien zu löschen, indem das entfernte Objekt zurück in einen Typ 1 umgeschaltet wird, wahrscheinlich um lokalen Speicher zu gewinnen oder den "Cache-Kohärenz-Überbau" zu verringern.

[0061] Wie in Fig. 7 gezeigt ist, können manche Objekte lokal zum Zwecke der Aktualisierung gespeichert werden; in solchen Situationen muß nur eine Kopie an irgendeinem Zeitpunkt veränderbar sein, und die anderen Objekte müssen in einem kohärenten Zustand gehalten werden. Zum Beispiel kann ein Verarbeitungsknoten, der eine Nutzerschnittstelle unterstützt in einen Befehlsstrom (Liste) schreiben, der für alle Verarbeitungsknoten kohärent sichtbar sein muß, die den Befehl implementieren könnten. Wie in Fig. 7 gezeigt ist, kann ein entferntes Typ 2-Objekt ein anderes entferntes Typ 2-Objekt bezeichnen (Blöcke **112** & **113** bezeichnen Block **119**). Das zweite entfernte Typ 2-Objekt kann wiederum eine lokale Speicheradresse (wie gezeigt) sein oder kann eine Knoten-ID und Speicheradresse verwenden (Blöcke **117** & **118**), um ein Objekt auf einem dritten Knoten (nicht gezeigt) zu bezeichnen.

[0062] Um die Kohärenz zu erhalten, muß die verteilte, kompilerorientierte Datenbank ein Datenbankobjekt mit der ersten nicht-lokalen Kopie verknüpfen, wie in Fig. 8 gezeigt ist. Da die meisten Objekte auf einem Knoten nur lokal referenziert werden, wäre es nicht platzeffizient, solche Information in alle Objekte in der Datenbank zu packen. (obwohl dies logisch korrekt wäre). Statt dessen können irgendwelche Mittel zum Lokalisieren des Knoten-Identifizierers der ersten nicht-lokalen Kopie, des Schreibstatus und der Speicheradresse (auf dem entfernten Knoten) auf dem Verarbeitungsknoten gehalten werden, dem das ursprüngliche Objekt gehört. Viele solcher Mittel sind Fachleuten bekannt, einschließlich vollständig assoziativer Hardware- oder Software-Hash-Tabellen.

[0063] Zahlreiche Cache-Speicherungs-Schemata sind Fachleuten bekannt. Das obige Schema beschreibt nur ein Mittel (eine bevorzugte Ausführungsform) des Implementierens kohärenter, verteilter Datenbanken zum Kompilieren. Viele andere Techniken könnten aus der Cache-Speicherungs-Literatur als Mittel zum Implementieren des Cache-Speicherungs-Mechanismus angepaßt werden, der von der Gesamterfindung benötigt wird.

[0064] Einige Objekte, wie vordefinierte Symbole, die von einer Sprache oder Umgebung definiert sind, Ganzzahl-Literale, Fließkomma-Literale, Buchstaben-Literale, String-Literale und andere kanonische Objekte haben gewöhnlich eine Darstellung pro Verarbeitungsknoten. Beispielsweise gibt es nur eine Darstellung der ganzen Zahl 3. Es gibt keine Notwendigkeit, solche Objekte für eine Aktualisierung kohärent zu halten. Somit kann die kompilerorientierte Datenbank optimiert werden, indem solche kanonischen Objekte, (mittels Wert) in die lokale Darstellung abgebildet werden, eher als ein nicht-lokales Speichern in einen Cache-Speicher durchzuführen. Solch ein Mittel ist in Fig. 9 gezeigt.

[0065] Mit der Zeit können Objekte in der Datenbank nicht länger erreichbar sein. Wo es möglich und sicher ist, ist es nützlich, alle in den Cache-Speicher kopierten Kopien solcher Objekte und das Originalobjekt zu löschen. Datenbankoperationen werden beschleunigt, indem entweder ein Referenzzähler oder andere Formen einer Müllsammlungsvorrichtung vorgehalten werden, die Fachleuten bekannt sind.

[0066] Fig. 10 stellt eine modifizierte Form eines Codeerzeugers (Block **44**), eines Verknüpfers/Laders (Block **45**) und eine Ausführung/Simulation (Block **46**) dar. Eher als in denselben Adreßraum wie die Klienten zu kompilieren, existiert das Ausführbare/die Simulation in einem getrennten Betriebssystemprozeß (Block **162**). Der Codeerzeuger, der Verknüpfer/Lader und die Ausführungs-/Simulationssteuerung lesen und schreiben den getrennten Simulations-/Ausführungs-Betriebssystem-Prozeß auf den Knoten über Mechanismen, die Fachleuten wohl bekannt sind, wie gemeinsamen Speicher, Zwischenprozeß-Kommunikation oder Betriebssystemnachrichten (gezeigt als **163**, **164**, **165**).

[0067] Die Trennung der kompilerorientierten Datenbank-Klienten von dem tatsächlich Ausführbaren hat mehrere kritische Vorteile. Erstens ist der gesamte virtuelle Adreßraum, der von dem Betriebssystem an jeden Prozeß geliefert wird, sowohl für die Datenbank als auch den Klienten (einen Prozeß) und die Simulation/das Ausführbare (einen weiteren Prozeß) verfügbar. Solch ein Aufteilen nimmt ältere Architekturen und Betriebssysteme auf, die auf 2 Millionen Byte virtuellen Speicherbereich beschränkt sind und vermeidet die Notwendigkeit für erweiterte Adreßdarstellungen auf neueren Architekturen und Betriebssystemen (die mit einem gesteigerten Speicherverbrauch verknüpft sind). Zweitens steigert die Aufteilung die Systemintegrität, da eine Fehl-Operation der Simulation/Ausführung nicht direkt den Code oder die Datenstrukturen ändern kann, die mit der Datenbank oder ihren Klienten verknüpft sind. Schließlich adressiert die Teilung Code oder Modellsicherheitsfragen, die auftreten können, falls eine Bibliothek oder ein Hardware-Modell-Bestandteil empfangen, entschlüsselt, kompiliert und mit einem anderen Programm oder einem Nutzercode verknüpft werden, die versuchen, einen direkten Zugriff auf die mehr abstrakte, kompilerorientierte Datenbankdarstellung zu erlangen (vielleicht zum Zwecke des Rückwärts-en-

gineering). Die im Betrieb befindliche geeignete Betriebssystemsicherung, der Dualprozeßansatz, der in Fig. 10 gezeigt ist, verkompliziert solches Rückwärts-engineering erheblich.

[0068] Nachdem die Prinzipien der Erfindung in der bevorzugten Ausführungsform dargestellt und beschrieben worden sind, sollte es für den Fachmann offensichtlich sein. daß die Erfindung in der Anordnung und im Detail modifiziert werden kann, ohne von solchen Prinzipien abzuweichen. Wir beanspruchen alle Modifizierungen, die innerhalb des Bereichs der folgenden Ansprüche fallen.

Patentansprüche

1. Parallelprozessorsystem mit wenigstens einem Knoten, welches programmiert ist, um einen globalen Zugriff mehrerer Klienten auf eine verteilte, kompilerorientierte Datenbank zu liefern, wobei der globale Zugriff eine gleichzeitig betreibbare Kompilierungs-, Simulations- und/oder Softwareausführungsbetriebsart umfaßt und wobei das Parallelprozessorsystem die folgenden Merkmale aufweist: Wenigstens zwei Prozessoren; einen Speicher, der mit wenigstens einem Prozessor betreibbar verbunden ist, wobei der Speicher ein lokaler Speicher oder ein gemeinsamer Speicher ist; und Zwischenverbindungsmechanismen zum Verbinden der wenigstens zwei Prozessoren mit dem Speicher; wobei die mehreren Klienten eine oder mehrere der folgenden Funktionen für Hilfsrepräsentations-Datenbankobjekte in der verteilten, kompilerorientierten Datenbank ausführen: Inkrementalanalyse eines Quellcodes in die Datenbank; Inkrementalausführung und/oder Inkrementalreihung von vorher analysierten Informationen in die Datenbank, um neue oder überarbeitete Datenbankeinträge zu liefern; Inkrementaloptimierung, welche Datenbank-Inhalte transformiert, um eine effizientere Simulation, eine effizientere Ausführung oder eine besser beobachtbare Ausführung zu liefern; Inkrementalcodeerzeugung, -zusammenbau und -verbinden, was Datenbankinhalte transformiert, um Repräsentationen zu liefern, die auf einer Kombination programmierbarer Prozessoren und rekonfigurierbarer Logikelemente direkt ausführbar ist; Laufzeit-, I/O- und Filesystem-Operationen, welche Planung, Kommunikation und Rückrufe innerhalb und unter direkt ausführbaren Simulationen/Ausführungen liefern; Testen, was für interaktives Beginnen, Stoppen und Testen des Zustands der Simulation/Ausführung sorgt; Profilieren oder Ereignis protokollieren der Simulation/Ausführung, um Simulations-/Ausführungsergebnisse für äußere Werkzeuge oder Nutzer zugreifbar zu machen; und graphisch interaktive Nutzerschnittstellenübertragungen, wobei die Datenbank kohärent Information zwischenspeichert, die auf dem wenigstens einen Knoten von Nutzen ist.

2. Parallelprozessorsystem nach Anspruch 1, dadurch gekennzeichnet, daß die wenigstens zwei Prozessoren jeweils eine rekonfigurierbare Instruktionsfolge-Architektur unterstützen, wobei wenigstens einer der wenigstens zwei Prozessoren mit rekonfigurierbaren logischen Ausführungspipelines erweitert ist.

3. Parallelprozessorsystem nach Anspruch 1, dadurch gekennzeichnet, daß wenigstens einer der wenigstens zwei Prozessoren mit rekonfigurierbaren logischen Ausführungspipelines erweitert ist, die mittels Agieren eines Registers zum Registrieren einer Instruktion aus einem Instruktionsstrom wenigstens eines der wenigstens zwei Prozessoren geliefert werden.

4. Parallelprozessorsystem nach Anspruch 1, dadurch gekennzeichnet, daß wenigstens einer der wenigstens zwei Prozessoren mit rekonfigurierbaren logischen Ausführungspipelines erweitert ist, die mittels Agieren von Instruktionen geliefert werden, die von dem Instruktionsstrom der wenigstens zwei Prozessoren auf eine feste Adresse abgebildet ist.

5. Parallelprozessorsystem nach einem der vorangehenden Ansprüche, dadurch gekennzeichnet, daß der Klient zwischen wenigstens zwei kommunizierenden Betriebssystemprozessoren arbeitet, um zusätzlichen adressierbaren Speicher und Softwareschutz zu liefern und um eine Zielsoftware-Ausführungsumgebung zu emulieren.

6. Parallelprozessorsystem nach einem der vorangehenden Ansprüche, dadurch gekennzeichnet, daß die mehreren Klienten schrittweise Quellcode mittels schrittweiser und kohärenter Änderungen in der Datenbank ändern, um eine schrittweise geänderte und neu zugeordnete Simulation/Ausführung zwischen den Prozessoren und rekonfigurierte logische Einrichtungen zu erreichen, wobei die Änderungen des Quellcodes einen Simulations-/Ausführungszustand aufrechterhalten, um sich entwickelnde Anforderungen für eine Unterbrechung der Simulation/Ausführung zu erfüllen.

7. Parallelprozessorsystem nach einem der vorangehenden Ansprüche, dadurch gekennzeichnet, daß das Unterbrechen einer Simulation/Ausführung Aktionen aus der folgenden Gruppe umfaßt: Triggern von Rückrufen, Rückrufe, Substituieren von Ereignis getriebenen/Zyklus getriebenen/kontinuierlichen Domain-Algorithmen oder Erzeugen eines Zwischen-Simulations-/Ausführungszustands, der für ein detaillierteres Testen, Anzeigen oder Aufreihen notwendig ist.

8. Parallelprozessorsystem nach einem der vorangehenden Ansprüche, dadurch gekennzeichnet, daß Beziehungsinformation, die in der Datenbank bezogene Information, die in einem Knoten vorhan-

den ist, und Information bezieht, die nur in einem entfernten Knoten präsent ist, mittels eines entfernten Proxy-Datensatzes repräsentiert wird, wobei der entfernte Proxy-Datensatz eine kohärente Zwischenspeicherung der Information in dem entfernten Knoten liefern kann, indem Speicherverfügbarkeit und Speicherkohärenz genutzt werden, um eine zwischengespeicherte Kopierlebensdauer zu begrenzen.

9. Parallelprozessorsystem nach einem der vorangehenden Ansprüche, dadurch gekennzeichnet, daß Analyse-, Ausarbeitungs- und Laufzeitabhängigkeits-Beziehungen zwischen Datenbankinformation eine Kompilierungs-, eine Simulations- oder eine Ausführungs-Klientaktivität auf demselben oder anderen Knoten triggern können, so daß die Datenbank den Trigger und die zugehörige Kompilierung, Simulation oder Ausführung unabhängig vom Ort der Information und von Klienten auf verschiedenen Knoten und von Komponenten auf dem Knoten koppelt.

10. Parallelprozessorsystem nach einem der vorangehenden Ansprüche, dadurch gekennzeichnet, daß die Ausarbeiter-Klientoperation vor einer Codeerzeugung ausgearbeitete Datenbankobjekte erzeugen kann, wobei die Datenbankobjekte vom Ausarbeiten wiedergenutzter Datenbankobjekte resultieren, welche aus einer Operation des Analysierer-Klienten oder von Objekten stammen, die durch eine vorhergehende Operation des Ausarbeiter-Klienten erzeugt werden.

11. Parallelprozessorsystem nach einem der vorangehenden Ansprüche, dadurch gekennzeichnet, daß die Ergebnisse der Kompilierung für eine folgende Konfiguration von programmierbaren Prozessoren und/oder Logikelementen speicherbar sind, um eine Simulation und/oder eine Programmausführung auszuführen.

12. Parallelprozessorsystem nach einem der vorangehenden Ansprüche, dadurch gekennzeichnet, daß auf die Ergebnisse der Simulation oder der Ausführung von zwei oder mehreren äußeren Werkzeugen oder Nutzern gleichzeitig zugegriffen werden kann.

Es folgen 10 Blatt Zeichnungen

Anhängende Zeichnungen

FIG. 1

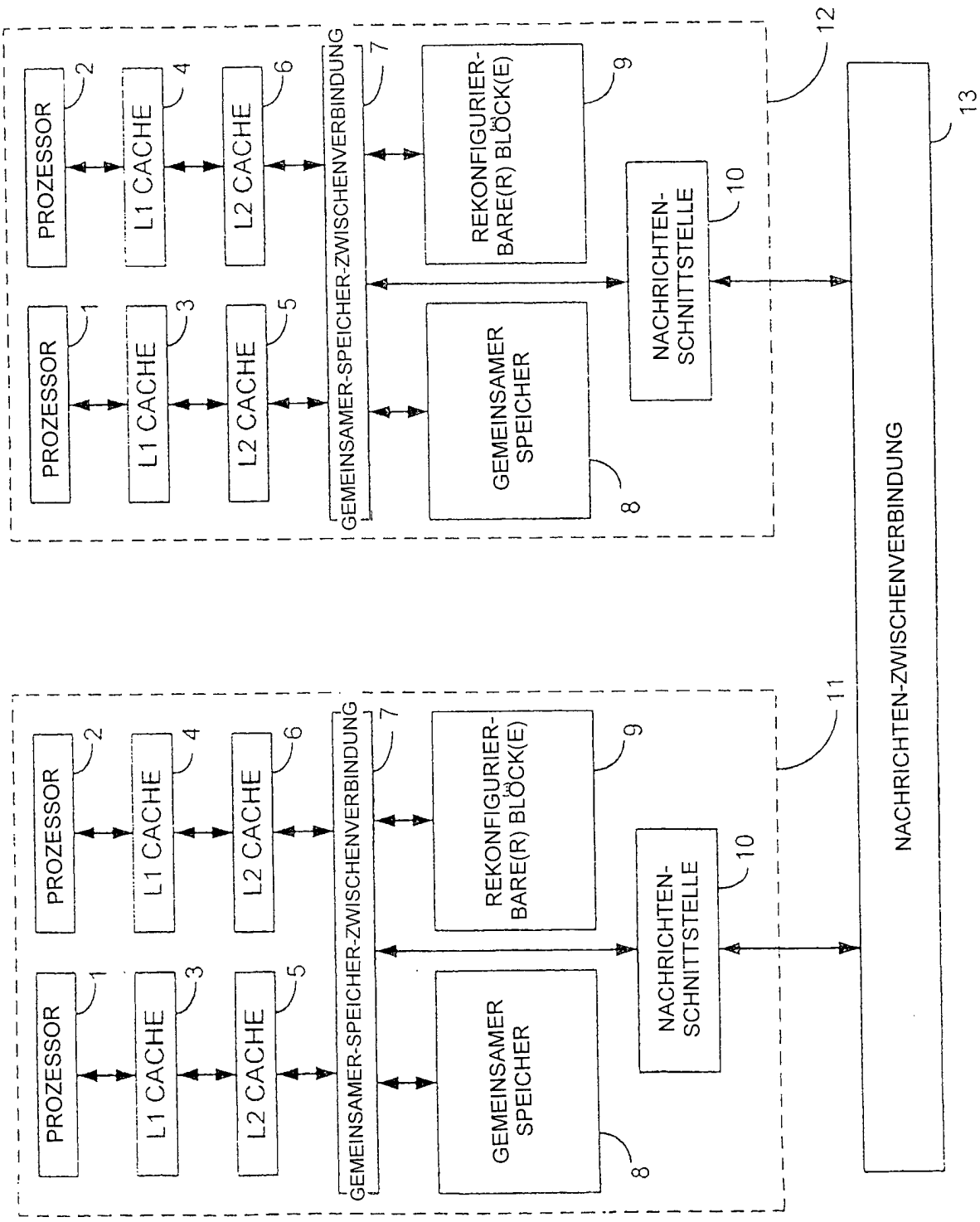
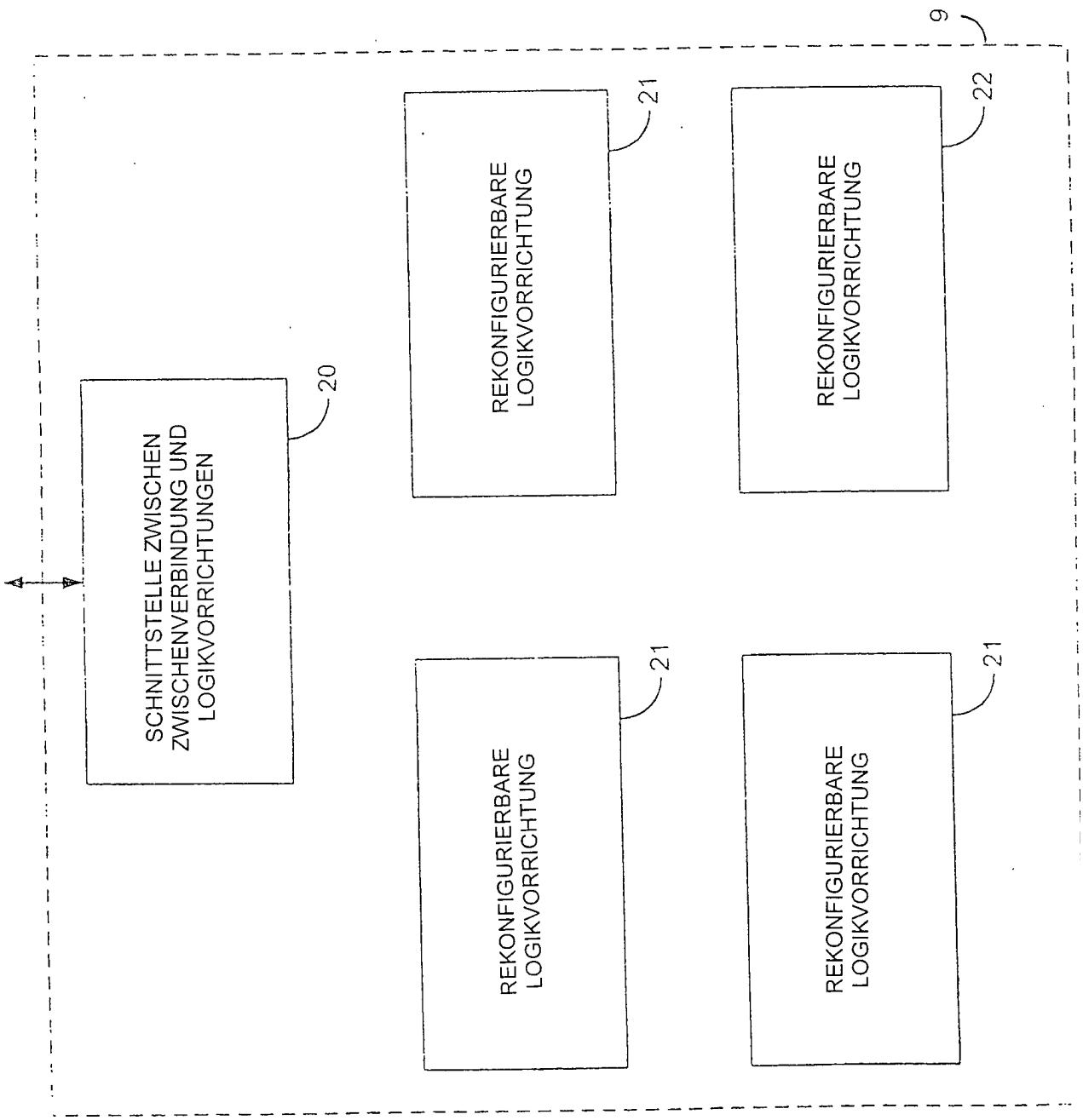


FIG. 2


9
SCHNITTSTELLE ZWISCHEN ZWISCHENVERBINDUNG UND LOGIKVORRICHTUNGEN
20
REKONFIGURIERBARE LOGIKVORRICHTUNG
21
REKONFIGURIERBARE LOGIKVORRICHTUNG
22
REKONFIGURIERBARE LOGIKVORRICHTUNG
21
REKONFIGURIERBARE LOGIKVORRICHTUNG
21

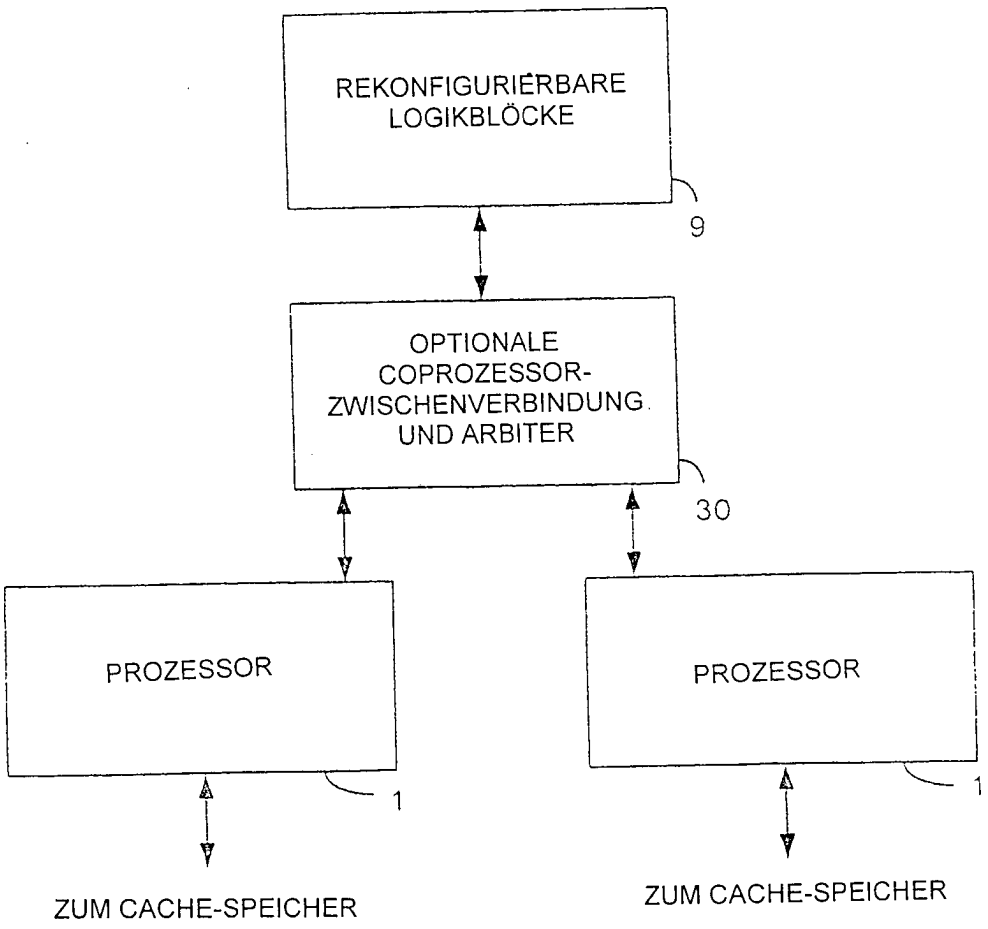
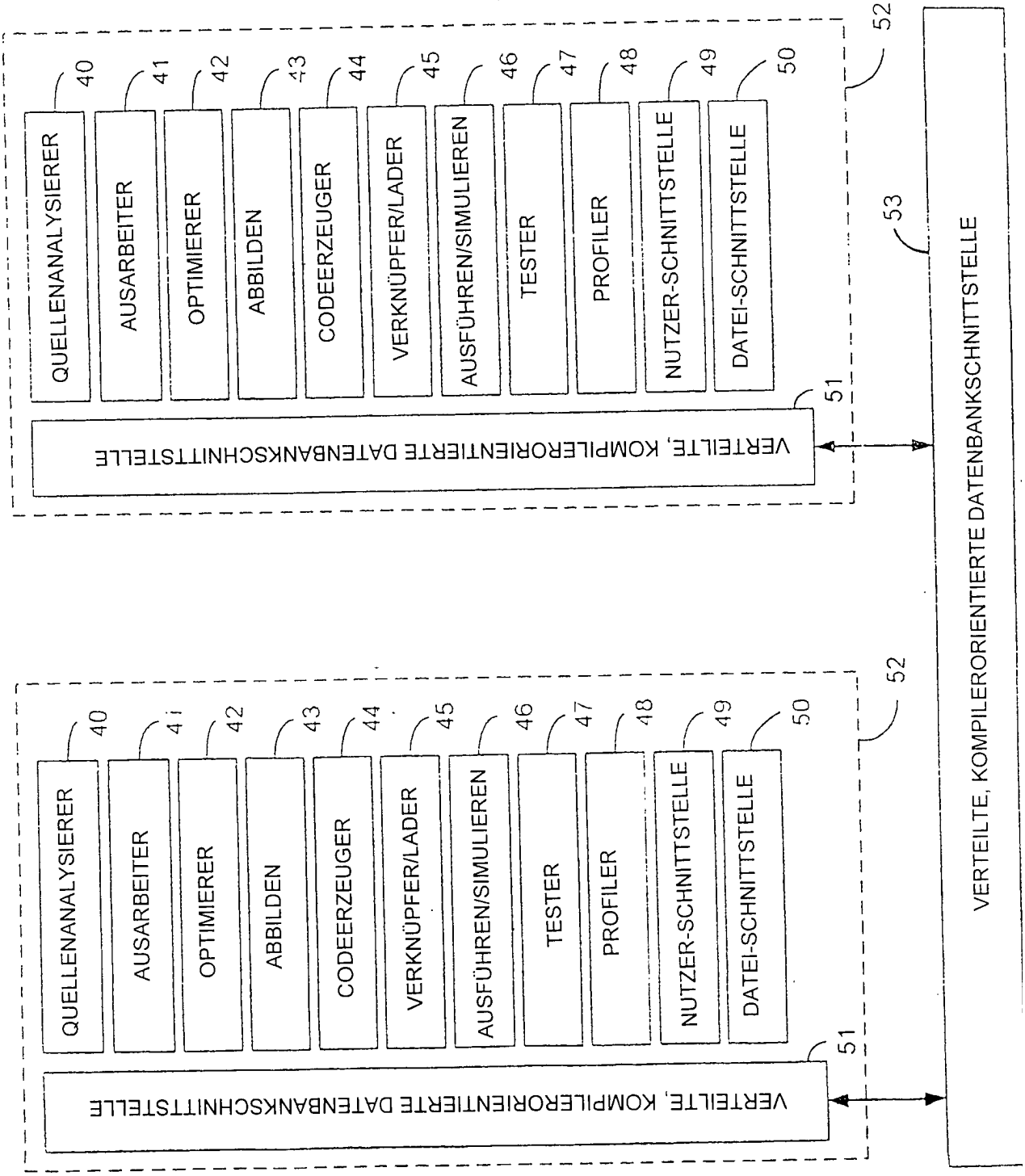

REKONFIGURIERBARE LOGIKBLÖCKE
9
OPTIONALE COPROZESSOR-ZWISCHENVERBINDUNG UND ARBITER
30
PROZESSOR
1
PROZESSOR
1
ZUM CACHE-SPEICHER
ZUM CACHE-SPEICHER

FIG. 3

FIG. 4


QUELLENANALYSIERER
AUSARBEITER
OPTIMIERER
ABBILDEN
CODEERZEUGER
VERKNÜPFER/LADER
AUSFÜHREN/SIMULIEREN
TESTER
PROFILER
NUTZER-SCHNITTSTELLE
DATEI-SCHNITTSTELLE
VERTEILTE, KOMPILERORIENTIERTE DATENBANKSCHNITTSTELLE
VERTEILTE, KOMPILERORIENTIERTE DATENBANKSCHNITTSTELLE
VERTEILTE, KOMPILERORIENTIERTE DATENBANKSCHNITTSTELLE
40
41
42
43
44
45
46
47
48
49
50
51
52
53

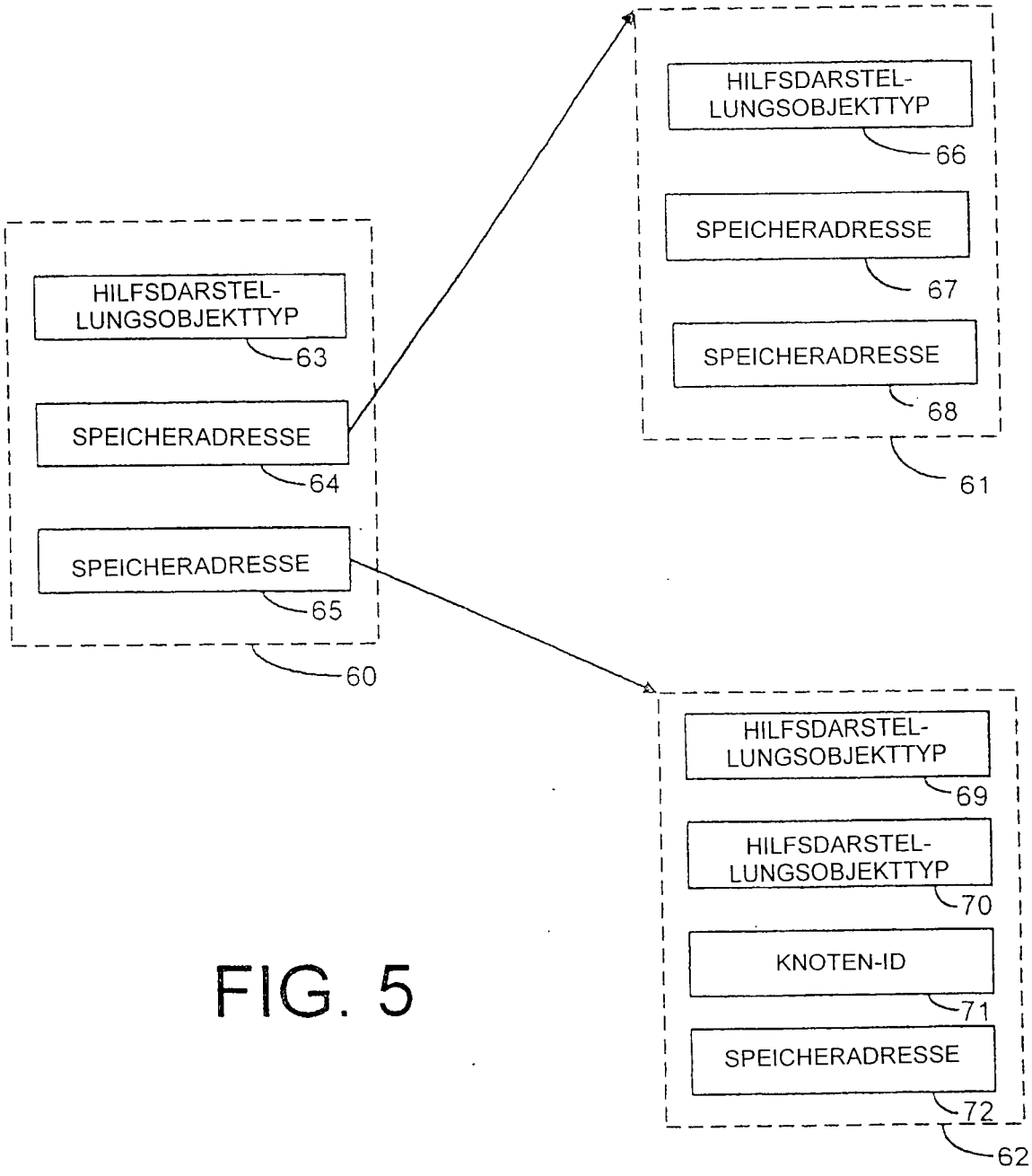

HILFSDARSTEL-
LUNGSOBJEKTTYP
63
SPEICHERADRESSE
64
SPEICHERADRESSE
65
60
HILFSDARSTEL-
LUNGSOBJEKTTYP
66
SPEICHERADRESSE
67
SPEICHERADRESSE
68
61
HILFSDARSTEL-
LUNGSOBJEKTTYP
69
HILFSDARSTEL-
LUNGSOBJEKTTYP
70
KNOTEN-ID
71
SPEICHERADRESSE
72
62

FIG. 5

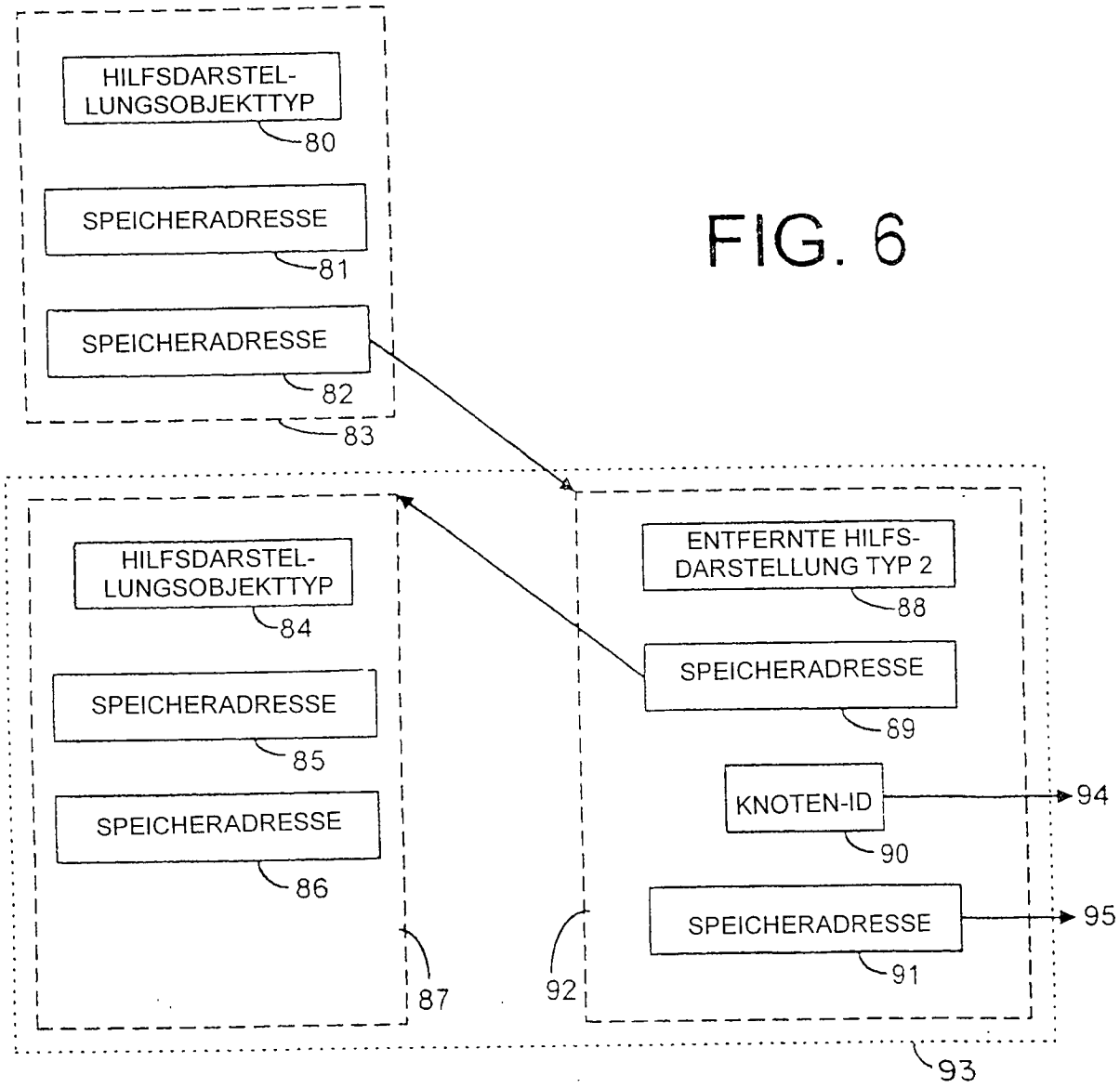

HILFSDARSTEL-
LUNGSOBJEKTTYP
80
SPEICHERADRESSE
81
SPEICHERADRESSE
82
83
HILFSDARSTEL-
LUNGSOBJEKTTYP
84
SPEICHERADRESSE
85
SPEICHERADRESSE
86
87
ENTFERNTE HILFS-
DARSTELLUNG TYP 2
88
SPEICHERADRESSE
89
KNOTEN-ID
90
94
SPEICHERADRESSE
91
95
92
93

FIG. 6

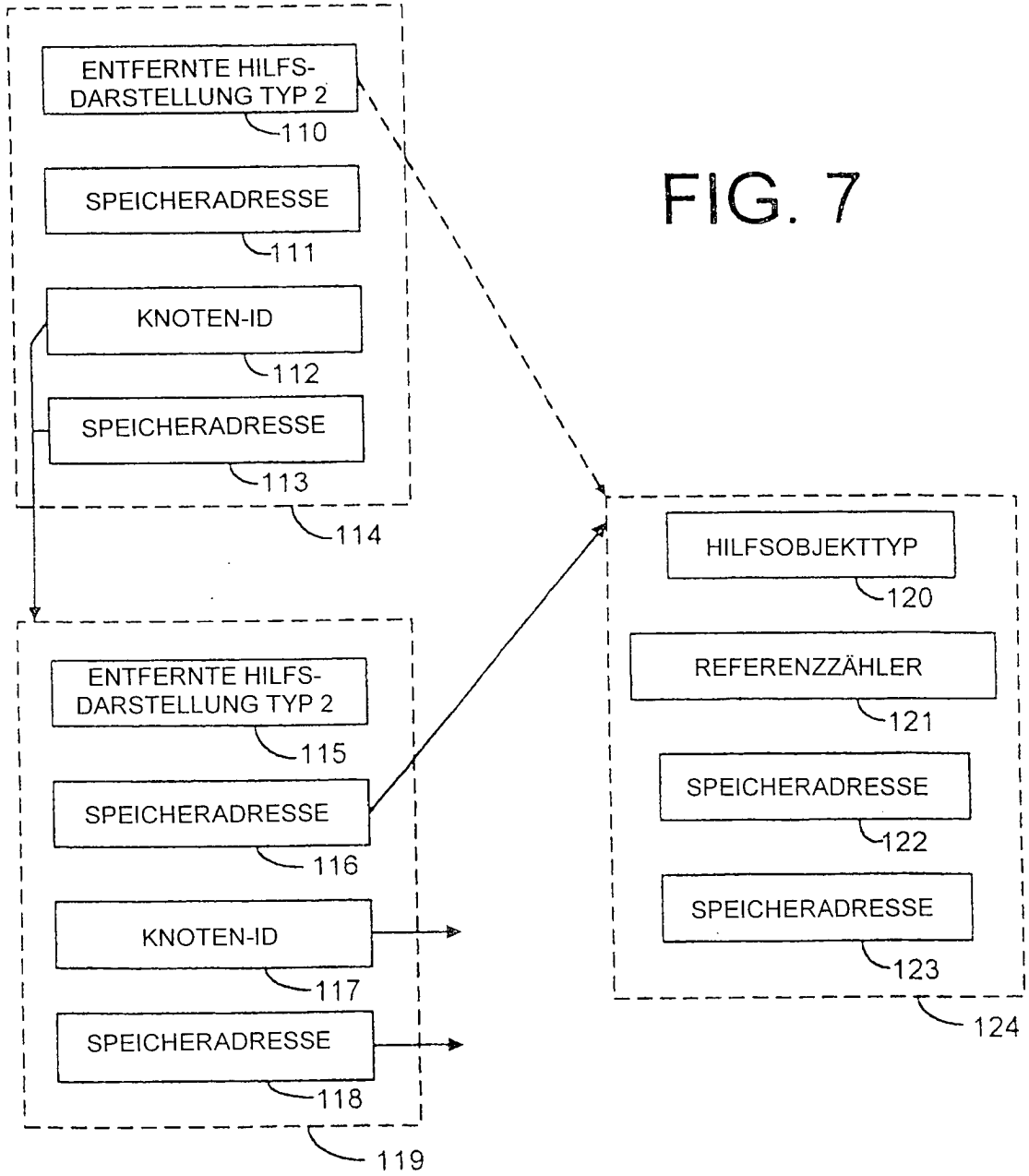

ENTFERNTE HILFS-DARSTELLUNG TYP 2
110
SPEICHERADRESSE
111
KNOTEN-ID
112
SPEICHERADRESSE
113
114
ENTFERNTE HILFS-DARSTELLUNG TYP 2
115
SPEICHERADRESSE
116
KNOTEN-ID
117
SPEICHERADRESSE
118
119
HILFSOBJEKTTYP
120
REFERENZZÄHLER
121
SPEICHERADRESSE
122
SPEICHERADRESSE
123
124

FIG. 7

FIG. 8

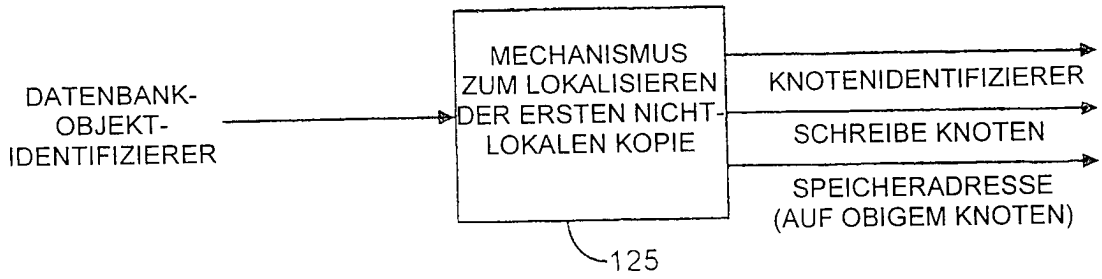

DATENBANK-
OBJEKT-
IDENTIFIZIERER
MECHANISMUS
ZUM LOKALISIEREN
DER ERSTEN NICHT-
LOKALEN KOPIE
125
KNOTENIDENTIFIZIERER
SCHREIBE KNOTEN
SPEICHERADRESSE
(AUF OBIGEM KNOTEN)

FIG. 9

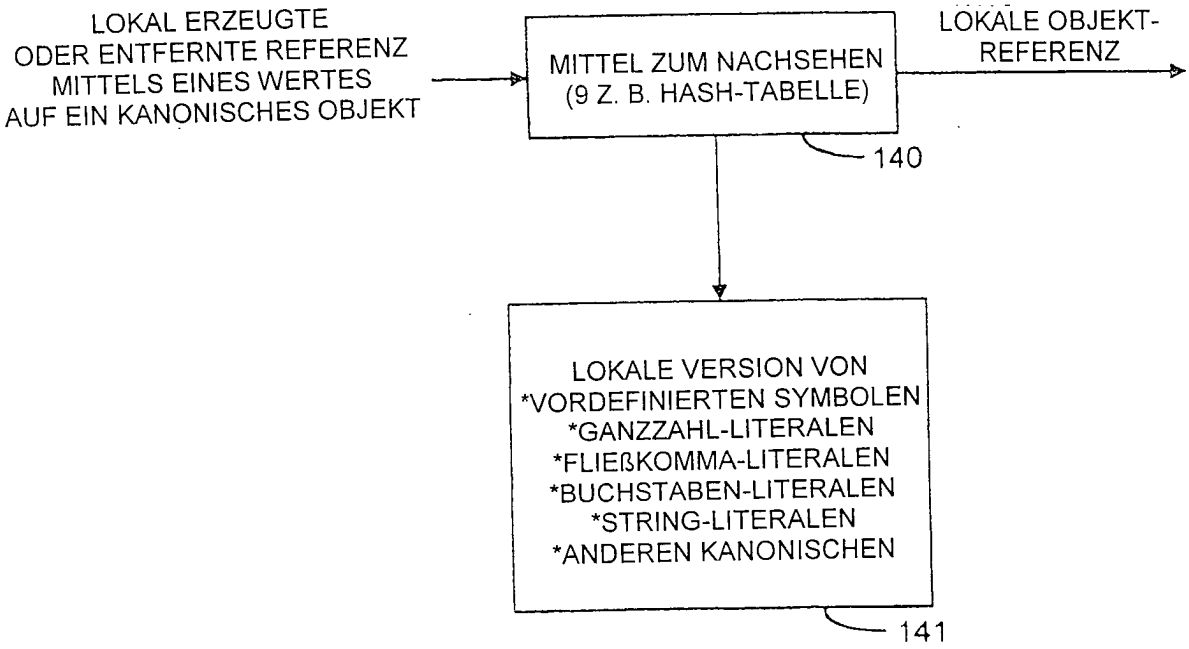
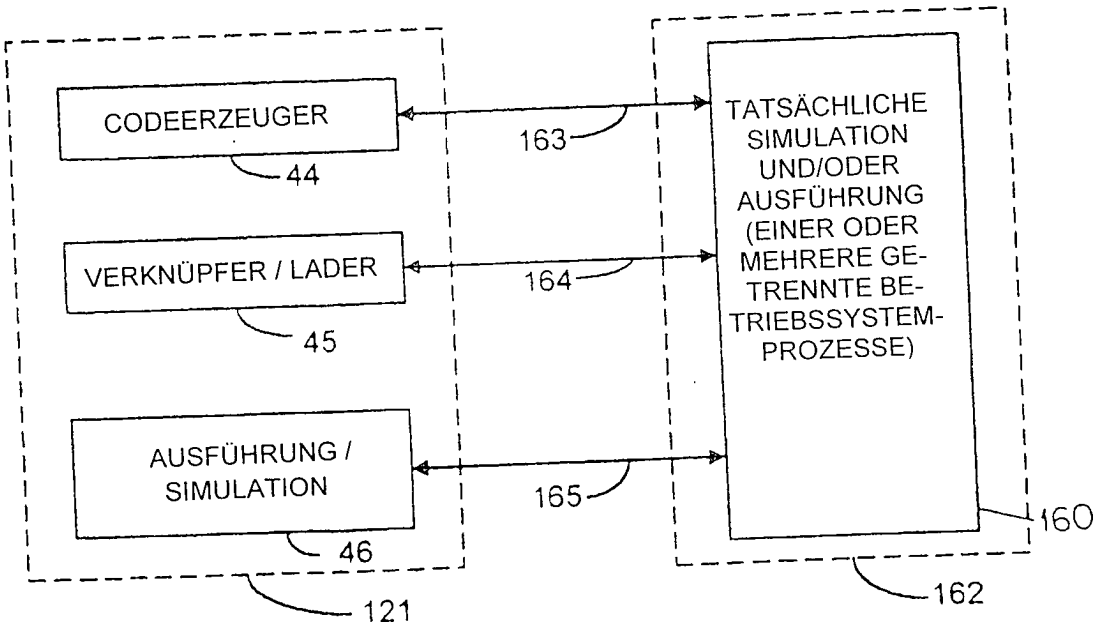

LOKAL ERZEUGTE
ODER ENTFERNTE REFERENZ
MITTELS EINES WERTES
AUF EIN KANONISCHES OBJEKT
MITTEL ZUM NACHSEHEN
(9 Z. B. HASH-TABELLE)
LOKALE OBJEKT-
REFERENZ
140
LOKALE VERSION VON
*VORDEFINIERTEN SYMBOLEN
*GANZZAHL-LITERALEN
*FLIEßKOMMA-LITERALEN
*BUCHSTABEN-LITERALEN
*STRING-LITERALEN
*ANDEREN KANONISCHEN
141

FIG. 10


CODEERZEUGER
44
VERKNÜPFER / LADER
45
AUSFÜHRUNG / SIMULATION
46
121
163
164
165
TATSÄCHLICHE SIMULATION UND/ODER AUSFÜHRUNG (EINER ODER MEHRERE GETRENNTE BETRIEBSSYSTEMPROZESSE)
160
162