Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

1.1.1 Memory Bottleneck

Eine wesentliche Bedingung, effiziente Anwendungen zu programmieren, besteht darin, dass die Register des Prozessors ausreichend schnell mit Daten versorgt werden können. Die Geschwindigkeiten aktueller Prozessoren betragen inzwischen einige GHz; die Zugriffszeit auf den Hauptspeicher (DRAM) beträgt z. Z. um die 70 ns. Bei einem Prozessor mit einen Takt von 2 GHz und einem Hauptspeicher mit einer Zugriffszeit von 70 ns würde ein Speicherzugriff 140 Taktzyklen dauern. Auf dieser Grundlage ist es schwierig, leistungsfähige Programme zu schreiben. Andererseits wäre ein Speicher, der aus schnellen SRAM-Bausteinen bestehen würde, zu groß, würde zuviel Energie verbrauchen und wäre vor allem viel zu teuer. Die Geschwindigkeit eines Speicherzugriffs ist nicht nur viel geringer als die des Prozessors; der Abstand zwischen beiden Geschwindigkeiten nimmt auch noch logarithmisch zu. Die Situation ist in Abbildung 1.1 dargestellt. Die Kluft zwischen Prozessor- und Speicherzugriffsgeschwindigkeit ist unter den Namen *memory bottleneck* ein wohlbekanntes Problem.

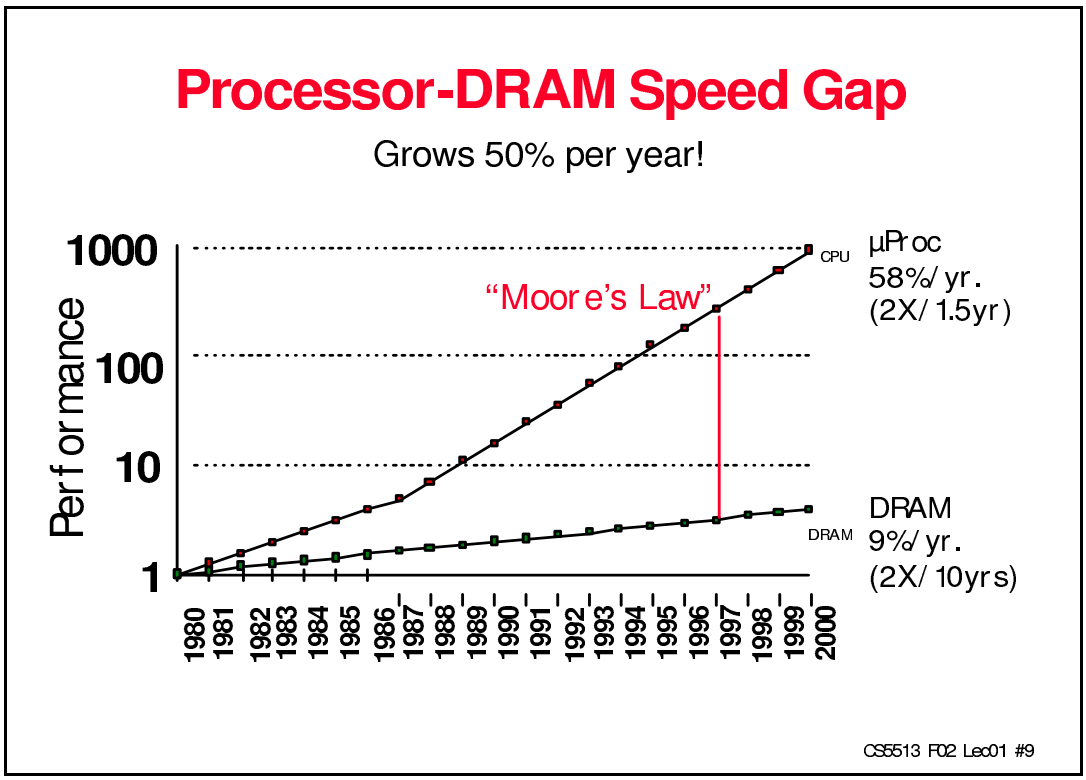


Abbildung 1.1: Die Geschwindigkeiten von DRAM und CPU im Verlauf der Jahre. „Moores Law“ besagt, dass sich die Prozessorleistung alle 18 Monate verdoppelt. Man beachte die logarithmische Skala, die das wirkliche Auseinanderfallen zwischen den beiden Geschwindigkeiten stark verharmlost darstellt. Es nimmt ebenfalls exponentiell zu.

1.1.2 Das Prinzip der Lokalität

Die Lage scheint also nicht sehr erfreulich zu sein. Dass die Verhältnisse weit weniger düster sind, als sie scheinen, ist folgender Beobachtung zu verdanken: Ein Programm greift sehr ungleichmäßig auf Daten und Code seines Adressraums zu. Grob gesagt verbringt es 90% seiner Ausführungszeit in 10% seines Codes. Konsequenterweise wird dieses Verhalten eines Executables als *90/10-Regel* bezeichnet. Gewisse Abschnitte eines Programms werden also weit häufiger durchlaufen als andere, und in Folge dessen wird auf die zugehörigen Speicherbereiche weit häufiger zugegriffen, als auf andere. Dies schlägt sich auch im sogenannten *Prinzip der Lokalität* nieder. Es zerfällt in zwei Unterprinzipien:

zeitliche Lokalität: Das *Prinzip der zeitlichen Lokalität* besagt, dass wenn auf ein Datum zugegriffen wird, mit hoher Wahrscheinlichkeit in naher Zukunft wieder darauf zugegriffen wird. Dies liegt vor allem daran, dass etwa 90% der Ausführungszeit eines Programms in Schleifen zugebracht wird.

räumliche Lokalität: Das *Prinzip der räumlichen Lokalität* besagt, dass wenn auf ein Datum zugegriffen wird, mit hoher Wahrscheinlichkeit naher Zukunft auch auf die benachbarten Daten zugegriffen wird. Dies ist z. B. der Fall, wenn in einer Schleife ein Index inkrementiert und auf die Elemente eines Arrays zugegriffen wird.

Anders ausgedrückt, konzentrieren sich die Speicherreferenzen eines Prozesses immer nur auf einen schmalen Ausschnitt seines Adressraums. Eine Linderung des obengenannten Problems besteht also darin, dass ein kleiner, schneller Speicher eine Kopie eines Ausschnittes des RAM enthält und die meisten Speicherreferenzen Daten dieses Ausschnitts betreffen. Nur wenn die referenzierten Speicherzellen dort nicht gefunden werden können, wird direkt auf den RAM zugegriffen. Dabei werden die betreffenden Daten auch automatisch in diesem kleinen, schnellen Speicher geschrieben. Die verdrängten Daten müssen eventuell in den RAM zurückgeschrieben werden. Insgesamt enthält dieser Speicher also immer einen ständig aktualisierten lokalen Ausschnitts des RAMs, der die meisten Speicherrefernzen der Instuktionen bedienen kann. Der Prozessor erhält dadurch die Illusion eines großen und schnellen Speichers. Dieser Speicher wird als *Cache* bezeichnet.

1.1.3 Zunehmender Abstand

Unglücklicherweise nimmt die Kluft zwischen Prozessor- und Speicherzugriffsgeschwindigkeit immer weiter zu. Zwar steigen beide exponentiell an, aber die eine verdoppelt sich alle 1 1/2 Jahre und die andere alle 10 Jahre. Um diesen wachsenden Abstand auszugleichen, muss die Leistung des Caches immer weiter gesteigert werden, d. h. es müssen immer mehr Datenzugriffe direkt vom Cache bedient werden können. Hierzu gibt es verschiedene Möglichkeiten: Die Prozessorhersteller bzw. die Rechnerarchitekten könnten z. B. größere Caches bauen. Der Anwendungsprogrammierer muss aber die technischen Gegebenheiten hinnehmen und kann nur versuchen, seinen Code zu verbessern, d. h. die Lokalität seines Programms zu optimieren. Dazu muss er sich ersteinmal ein klares Bild über das Speicherzugriffsverhalten seines Programms bei der Ausführug machen. Leider können die gegenwärtigen Performance Measurement Tools aber nur die Anzahl der verschiedenen

Cache-Events in einem bestimmten Programmabschnitt darstellen. Auf diese Weise können zwar die Stellen, die ein unbefriedigendes Speicherverhalten verursachen, ausfindig gemacht werden. Es ist aber nicht möglich, die Variablen, die für diese Cache-Events verantwortlich sind, zu bestimmen. Der Programmierer benötigt aber diese Variablen (in Form ihres Namens), um einen Bezug zum Quellcode seines Programms herstellen und dessen Datenzugriffe optimieren zu können.

1.2 Überblick

Im Folgenden wird gezeigt, wie dieses Konzept in der heutigen Speicherarchitektur realisiert ist und welche Gründe es für Cachefehlzugriffe geben kann (Kapitel 2). Darauf aufbauend wird im Kapitel 3 dargestellt, welche Methoden angewandt werden können, um den verschiedenen Arten dieser Cachefehlzugriffe zu begegnen und die Lokalität der Daten zu optimieren. Im Kapitel 4 werden Werkzeuge vorgestellt, mit denen die Performance eines Programms gemessen und analysiert werden kann und welche Mängel sie bei der Analyse des Speicherzugriffsverhalten einer Anwendung haben. Das Kapitel 7 beschreibt die Schnittstelle der data_mapping_lib und das Kapitel 6 die Details der Implementierung.

Kapitel 2

Caches

2.1 L1, L2 und L3 Caches

Ein 'Cache' ist ein kleiner, schneller Speicher, der dem Prozessor den jeweils aktuellen lokalen Ausschnitt des Hauptspeichers schnell zugänglich macht. Es ist in der Regel aus SRAM-Baussteinen aufgebaut. SRAM ist viel schneller als DRAM; hat aber die Nachteile, relativ groß zu sein und viel Strom zu verbrauchen, und ist auch noch sehr teuer. Deswegen muss ein Cache sehr klein seien (s. u.).
Üblicherweise gibt es zwei bis drei Cache-Ebenen, die L1-, L2- bzw. L3-Cache genannt werden. Nur das L1-Cache („first level cache“) tauscht direkt Daten mit den Registern des Prozessors aus. Mit wachsender Entfernung vom Prozessor steigt ihre Kapazität und sinkt die Zugriffsgeschwindigkeit pro Byte. Das L1-Cache befindet sich auf dem Prozessorchip; das L2-Cache auch oder auch nicht; das L3-Cache nicht. Das L1-Cache ist meistens in ein Instruktions-Cache und in ein Daten-Cache gespalten. Auf das eine wird nur lesend zugegriffen; auf das letztere lesend und schreibend.

2.2 Speicherhierachie

Unterhalb der Caches befindet sich der Hauptspeicher; unter dem Hauptspeicher der Hintergrundspeicher und unter dem Hintergrundspeicher der Archivspeicher. So ensteht eine Hierarchie von Speicherebenen mit dem L1-Cache an der Spitze und dem Archivspeicher als Basis. Sie ist durch folgende Eigenschaften gekennzeichnet:

1. Die Zugriffsgeschwindigkeit auf die Daten ist bei einem Speicher einer höheren Ebene größer als bei dem der darunterliegenden. Seine Kapazität ist dagegen geringer.

2. Nur benachbarte Ebenen tauschen direkt Daten miteinander aus. wenn z. B. bei einem zweistufigem Cache der Prozessor auf ein Datum zugreifen will, das es nur im Hauptspeicher gibt, dann muss es vom RAM zum L2-Cache kopiert werden, vom L2-Cache ins L1-Cache und von dort erst in ein Register des Prozessors.

3. Die kleinste Einheit von Daten, die zwischen zwei Ebenen ausgetauscht werden kann, wird ein 'Block' genannt. Im Falle des Caches wird sie auch 'Cacheline' genannt. Die Daten eines Blocks sind in einer Ebene also entweder komplett vorhanden oder sie fehlen komplett.

4. Die Blöcke einer höhere Ebene sind eine Kopie von Blöcken der darunterliegenden Ebene. Falls in ein Cache kopiert wird, erfolgt sie automatisch durch die Hardware. Eine Kopie vom Hintergrundspeicher in den RAM wird dagegen vom Betriebssystem veranlasst. Solange keine Kopie erfolgt ist, ist eine Cacheline nicht gültig.

5. Solange nur lesender Zugriff auf einen Block einer höheren Ebene erfolgt, ist er identisch mit dem korrespondierenden Block der darunterliegenden Ebene. Falls schreibender Zugiff auf ihn stattfindet, muss der korrespondierende Block der darunterliegenden Ebene irgendwann upgedated werden - spätestens wenn der Block der höheren Ebene verdrängt wird.

Da ein Cache möglichst schnell sein soll, wird seine Verwaltung von der Hardware übernommen. Im Gegensatz dazu wird der vituelle Speicher vom Betriebssystem verwaltet (also die Abbildung von virtueller in physische Adresse, das Einblenden einer virtuellen Seite in den Speicher und das eventuelle Auslagern von Seiten vom Hauptspeicher in einen Swap-Bereich). Ein Modell der Speicherhierachie und ihrer Wechselwirkung mit der CPU ist in Abbildung 2.1 zu sehen.

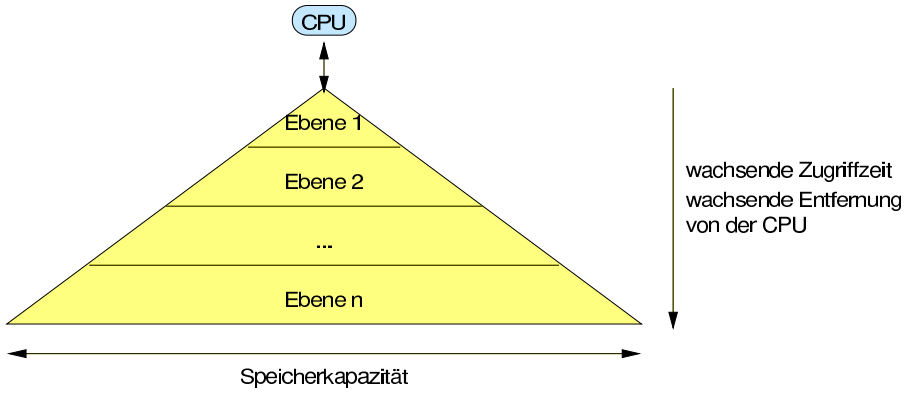


Abbildung 2.1: Das Modell einer Speicherhierachie.

2.3 Assiozativität

Ein Cache zerfällt in verschiedene Zeilen, die jeweils einen Block speichern können. Im allgemeinen kann ein bestimmter Block aber nicht jeder dieser Zeilen gespeichrt werden ;welche davon in Frage kommen, hängt von der Adresse des Blocks ab. Die Zahl der Zeilen, in der ein Block gespeichert werden kann, nennt man seine **Assiozativität**.
Im einfachsten Fall kann ein Block nur auf eine Zeile abbgebildet werden (die Assiozativität beträgt eins). Man spricht dann von einem **direct mapped** (auch **einfach assiozativen**) Cache. Die Adressbestimmung ist hier sehr einfach und geht deshalb schnell. Der Nachteil ist, dass sich häufig benötigte Blöcke, die auf die selbe Cachezeile abbgebildet werden, einander abwechselnd verdrängen. Deshalb verwendet man – vor allem bei dem L1-Cache – auch höhere Assiozativitäten. Wenn ein Block auf n verschiedene Zeilen abgebildet werden kann, nennt man das Cache **n-assiozativ**. Ist n gleich der Anzahl aller Zeilen des Caches, nennt man das Cache auch **voll-assiozativ**.

2.4 Das Auffinden eines Datums

Wenn der Prozessor lesend oder schreibend auf ein Datum zugreifen will, sieht er erst nach, ob es im Cache vorhanden ist. Dazu wird die Adresse des Datums dreigeteilt. Die mittleren Bits geben an, in welcher Zeile (im Falle eines direct mapped Caches) bzw. in welchen Zeilen (im Falle eines mehrfach assio-

zativen Caches) der Block gespeichert sein kann. Zu jeder Cacheline gehören ein ‘valid’-bit, welches anzeigt, ob auf die Cacheline überhaupt schreibend zugegriffen wurde. Es muss gesetzt sein, wenn die Cacheline das gesuchte Datum enthalten soll. Da viele Blöcke des Hauptspeichers auf dieselbe Cacheline abgebidet werden, muss es möglich sein zu bestimmen, ob die Daten der Cacheline zu der gesuchten Adresse gehört. Dazu müssen beim Laden einer Cacheline – zusätzlich zu den eigentlichen Daten – die höherwertigen Bits der Adresse gespeichert werden. Sie werden als *tag* bezeichnet. Wenn jetzt das bzw. ein in Frage kommendes tag mit den höheren Bits der gesuchten Adresse übereinstimmt, liegt ein ‘cache hit’ vor. Die niederwertigsten Bits geben den Offset eines Wortes innerhalb der Cacheline an. In Abbildung 2.2 ist der gesammte Vorgang nochmal zusammengefasst.

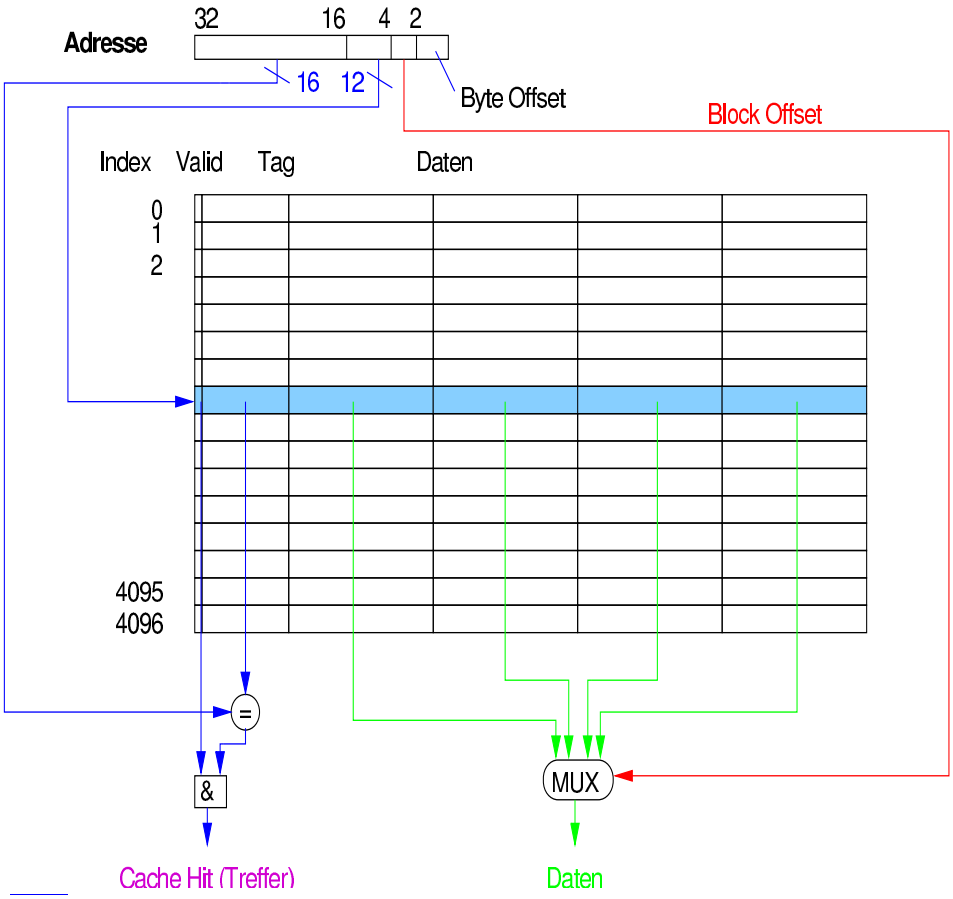


Abbildung 2.2: Das Suchen nach einer Adresse im Cache ist hier im Fall eines *direct mapped caches* dargestellt.

2.5 Aktualisierungsstrategien

Wenn der Prozessor schreibend auf eine Cacheline zugreift, muss das Datum auch im Hauptspeicher aktualisiert werden. Man nennt das Problem, den Hauptspeicher mit dem korrespondierenden Auschnitt im Cache in Einklang zu bringen, Konsistenzproblem: Hierfür gibt es zwei Strategien:

Zurückschreiben: Von Zurückschreiben (engl.: write back oder copy back) spricht man, wenn die Modifikation zunächst nur auf das Cache beschränkt bleibt und die Cacheline nicht sofort in den Hauptspeicher zurückgeschrieben wird. Erst wenn die Umstände es notwendig machen, z. B. wenn sie aus dem Cache verdrängt wird oder wenn bei SMP-Architekturen ein anderer Prozessor auf ihre Daten zugreifen will, wird die Cacheline in den Hauptspeicher zurückgeschrieben.

Durchschreiben: Von Durchschreiben (engl.: write through) spricht man, wenn bei einem schreibenden Zugriff auf das Cache die betreffende Cacheline sofort in den Hauptspeicher zurückgeschrieben wird.

2.6 Verdrängungsstrategien

Wenn ein Datum in das Cache geschrieben wird und alle in Frage kommenden Cachelines schon beschrieben sind, muss eine Cacheline verdrängt werden. Bei einem direct mapped cache gibt es hierfür nur eine Möglichkeit. Aber bei einem n-assiozativen Cache mit n>1 (insbesondere einem vollassiozativem Cache) stellt sich die Frage, welcher Block verdrängt werden soll. Hierfür sind viele Strategien möglich, verbreitet sind:

LRU: Ersetzt die Zeile, auf die in letzter Zeit am wenigsten häufig zugegriffen wurde. Schon von der Verwaltung des virtuellen Speichers her bekannte und bewährte Strategie. Es sind aber für jede Cachezeile zusätzliche Alterungsinformationen zu speichern und bei jedem Zugriff upzudaten. Außerdem müssen bei der Verdrängung einer Zeile eines n-assiozativem Caches i. a. n-1 Vergleiche durchgeführt werden.

Random: Die Zeile, die ersetzt wird, wird von einem (Pseudo-)Zufallszahlengenerator bestimmt. Schön einfach und deshalb schnell.

Wie alles beim Cache, werden diese Strategien von der Hardware ausgeführt.

2.7 Einige Definitionen

Um das Verhalten eines Caches zur Laufzeit eines Progamms zu beschreiben, sind folgende Bezeichnungen üblich:

cache hit: Ein *cache hit* (dt.: *Treffer*) liegt vor, wenn der Prozessor ein Datum, das er benötigt, im Cache vorfindet.

cache miss: Kein *cache hit* (dt.: *Fehlzugriff*)

hit time: Zeit, die benötigt wird, um festzustellen, ob ein Datum im Cache vorhanden ist und um es zu lesen oder zu schreiben. Wird i. f. mit t_h bezeichnet.

miss time: Zeit, die benötigt wird, um ein Datum, das im Cache nicht vorgefunden wird, zu lesen oder zu schreiben. Wird i.f. mit t_m bezeichnet.

miss penatity: Der Betrag, um der die *miss time* länger dauert, als die *hit time*.
$misspenality = t_m - t_h$

hit rate: Die Trefferrate, i. f. bezeichnet als p_h, wobei $0 \leq p_h \leq 1$

miss rate: Die Fehlzugriffsrate, i. f. bezeichnet als p_m, wobei $0 \leq p_m \leq 1$; $p_h + p_m = 1$

average access time: dt.: *mittlere (auch effektive) Speicherzugriffszeit*, kurz t_e. $t_e = p_h * t_h + p_m * t_m = t_h + p_m *$(miss penality)

2.8 Die Arten der „cache misses“

Es gibt drei verschiedene Gründe, weshalb es zu einem cache miss kommen kann:

compulsory miss: Wenn auf ein Datum zum ersten Mal während der Programmausführung zugegriffen wurde, kommt es zwangsläufig zu einem cache miss. Ein solcher cache miss wird deshalb auch *„compulsory miss“* genannt. Man spricht auch von einem „cold“ bzw. „upstart miss“.

capacity miss: Auch wenn intuitiv klar zu sein scheint, was unter der Anzahl der *„capacity misses“* zu verstehen ist, kann man es nur umständlich, nämlich auf dem Umweg eines vollassiozativen Caches, genau definieren. Dort ist die Anzahl der capacity misses gleich der Anzahl der cache misses minus der Anzahl der compulsory misses. Allgemein ist

die Anzahl der capacity misses gleich der Anzahl der capacity misses, die ein äquivalentes vollassiozatives Cache (d. h. eines von gleicher Speichergröße, gleicher Blockgröße und gleicher Ersetzungsstrategie) aufweisen würde.

conflict miss: Ein *„conflict miss“* liegt vor, wenn auf ein Datum zugegriffen wird, das sich schon Mal im Cache befunden hat, aber von einem anderen Datum verdrngt wurde. Ein conflict miss entsteht, weil im aktuellen Programmabschnitt zuviel Daten referenziert werden, die auf dasselbe Set des Caches abgebildet werden. Die Anzahl der conflict misses ist gleich der Anzahl der cache misses, minus der Anzahl der compulsory misses. minus der Anzahl der capacity misses. *„conflict misses“* werden auch als *„interference misses“* bezeichnet.

Kapitel 3

Methoden zur Optimierug der Datenlokalität

3.1 Allgemeines

Die verschiedenen Gründe für *cache misses* machen unterschiedliche Maßnahmen zu deren Beseitigung erforderlich. Z .B. können *compulsory misses* (und nur diese) durch *„data prefetching“* verhindert werden, also durch spekulatives Laden von Daten in das Cache. Die anderen beiden *cache misses* (*capacity* bzw. *conflict misses*) sind nur möglich, wenn sich die referenzierten Daten in einer Cacheline befinden, die sich schon mal im Cache befand, aber wieder verdrängt wurde. Man kann sie also vermeiden, indem man in anderen Abständen auf diese Daten zugreift oder sie in eine andere Cacheline schreibt. In beiden Fällen wird die Lokalität der Datenzugriffe verbessert. Hierzu gibt grundsätzlich zwei Möglichkeiten:

1. Man kann einfach nur die Zeilen des Sourcecodes umstellen. Das Ziel ist, dass benachbarte Anweisungen auf Daten zugreifen, die auch im Speicher benachbart abgelegt sind. In diesem Fall spricht man von *„Data Access Transformations“*. Voraussetzung ist natürlich, dass nichts an der Semantik des Programms verändert wird. Solche Transformationen nennt man auch *„legal“*.

2. Die andere Möglickeit, die Datenlokalität zu verbessern, besteht darin, das Layout der Daten zu verändern. Z. B. können die members eines structs so umgestellt werden, dass auf sie in derselben cacheline stehen, wenn ein member kurz nach dem anderen referenziert wird. In diesem Fall spricht man von *„Data Layout Transformations“*.

3.2 Abhängigskeitanalyse

Ein Programmierer wird vielleicht intuitiv sagen können, ob eine *„Data Access Transformation“* gültig ist oder nicht. Aber spätestens wenn die Optimierung der Lokalität automatisch vom Compiler vollzogen werden soll, braucht man hierfür genaue Regeln. Folgende Abhängigkeiten müßen bei der Umordnung der Zeilen eines Programms eingehalten werden:

1. Eine *„flow dependence“* (auch: *„real dependence“*) einer ersten Anweisung von einer zweiten besteht, wenn die erste Anweisung vor der zweiten ausgeführt wird und die erste Anweisung schreibend auf eine Variable zugreift, auf die die zweite lesend zugreift.

2. Eine *„anti dependence“* ist die Umkehrung einer „flow dependence“. Die erste Anweisung wird vor der zweiten ausgeführt und die erste Anweisung greift lesend auf eine Variable zu, auf die die zweite schreibend zugreift.

3. Eine *„output dependence“* zwischen zwei Anweisungen liegt vor, wenn die erste Anweisung vor der zweiten ausgeführt wird und beide Anweisungen schreibend auf eine Variable zugreifen.

4. Eine *„input dependence“* ist das Gegenteil einer „output dependence“. Die erste Anweisung wird vor der zweiten ausgeführt und beide Anweisungen greifen lesend auf eine Variable zu. Diese Abhnigkeit muss bei der Umordnung der Zeilen eines Programms nicht eingehalten werden, aber sie weist auf einen mehrfachen Zugriff auf dasselbe Datum hin. Der Compiler kann diesen Hinweis zur Optimierug der Datenlokalität benutzen.

3.3 Data Access Transformations

Wenn man einen Programmcode optimiert, wird man sich auf häufig benutzte Programmteile beschränken, d. h. auf Schleifen. Deshalb konzentrieren sich die Untersuchungen von „Data Access Transformations“ auf „Loop Transformations“, genauer gesagt auf vollkommen verschachtelte Schleifen. Das sind Schleifen, die alle nur eine weitere Schleife enthalten; mit der Ausnahme der innersten, die dann den Code enthält. Bei Schleifen gibt es zwei mögliche Gründe, weshalb eine Anweisung vor einer anderen ausgeführt werden kann:

- Die Indices der Schleifen sind bei beiden Anweisungen gleich, aber die Zeilennummer der ersten Anweisung ist kleiner als die der zweiten.

- Die Schleifenindices der ersten Anweisung sind kleiner als die bei der zweiten. Hier ist die Zeilenummer der beiden Anweisungen gleichgültig.

3.3.1 Loop Interchange

Manchmal ist es nicht wichtig, in welcher Reihenfolge (vollkommen) verschachtelte for-Schleifen ausgeführt werden. Nämlich dann, wenn zwischen allen Variablen im Inneren der Schleifen höchstens *output dependencies* bestehen. Wenn in diesem Fall zwei benachbarte Schleifen vertauscht werden, spricht man von *„loop interchange“*; wenn mehr als zwei oder nicht benachbarte Schleifen vertauscht werden, von *„loop permutation“*. Um den Nutzen der *loop interchange* für die Optimierug der Datenlokalität zu verstehen, soll erstmal der Begriff der *„stride“* definiert werden. Die *„stride“* ist der Abstand zwischen zwei Elementen eines Arrays im Speicher, auf die in zwei aufeinanderfolgenden Schleifendurchläufen zugegriffen wird. Ein Beispiel hierfür ist eine Matrix (ein 2-dimensionales Array), die im Speicher reihenweise gespeichert ist. Wenn in der Berechnung im Inneren der Schleifen spaltenweise auf die Elemente der Matrix zugegriffen wird, ist die *stride* gleich der Anzahl der Spalten der Matrix. Wenn reihenweise auf sie zugegriffen wird, ist die *stride* gleich eins. Durch *loop interchange* kann also eine Verkürzung der *stride* erreicht werden. Wenn ein Array-Element ein *cache miss* auslöst, werden ja mit seiner cacheline i. a. auch benachbarte Array-Elemente ins Cache geladen. Wenn dann auf diese zugegriffen wird, erniedrigt eine kürzere *stride* die Wahrscheinlichkeit, dass die zugehörige cacheline schon wieder aus dem Cache verdrängt wurde.

3.3.2 Loop Fusion

Von *„loop fusion“* spricht man, wenn zwei aufeinanderfolgende Schleifen, die dieselben Ober- und Untergrenzen und dasselbe Inkrement haben, zu einer Schleife zusammengefasst werden. Diese Transformation ist *legal*, wenn keine *flow-, anti-* oder *output-dependences* zwischen den Instruktionen der beiden Schleifen bestehen. Durch *loop fusion* kann die Anzahl der *capacity* und/oder *conflict misses* gesenkt und somit die Datenlokalität verbessert werden. Man stelle sich z. B. vor, dass beide Schleifen in der selben Reihenfolge auf die Elemente eines Arrays zugreifen, das als ganzes nicht im Cache Platz hat. Bei zwei getrennten Schleifen, wären die ersten Elemente des Arrays schon wieder aus dem Cache verdrängt, wenn die erste Schleife mit ihren Berechnungen fertig ist. Es kommt daher mit Sicherheit zu *cache misses*, wenn die zweite Schleife ihre Berechnungen startet und auf die ersten Elemente des Arrays

zugreifen will. Wenn beide Schleifen zu einer zusammengefasst werden, gibt es diese *cache misses* nicht.

3.3.3 Loop Tiling

Die Funktionsweise des *„loop tiling“* oder *„blockings“* macht mann sich am besten mit einem kleinen Beispiel klar. Gegeben seien zwei verschachtelte for-Schleifen, deren Berechnung im Inneren auf die Elemente einer Matrix zugreifen und die notwendigerweise eine große stride haben. Die Berechnung würde also notwendigerweise zu *capacity* und/oder *conflict misses* führen. Deshalb wird diese Matrix erst in verschiedene kleinere Untermatrixen gleicher Größe unterteilt. Sie sollen jetzt im Cache gespeichert werden können. Statt nun bei der Berechnung auf die Elemente der einen großen Matrix zuzugreifen, werden den zwei for-Schleifen zwei weitere äußere for-Schleifen vorgeschaltet, die die Untermatrixen der Reihe nach durchlaufen. Erst im Rumpf der inneren Schleifen wird dann auf die Elemente der Untermatrixen zugegriffen. So werden *capacity* bzw. *conflict misses* vermieden. Diese Lösung setzt natürlich voraus, dass die Reihenfolge, in der auf die Array-Elemente zugegriffen wird, entsprechend verändert werden kann.

3.4 Data Layout Transformations

Die bisherigen Techniken haben ein Verbesserung der Datenlokalität erreicht, indem sie die Reihenfolge, mit der ein Programm auf seine Daten zugreift, verändert haben. Leider können damit nicht alle Performanceengpässe beseitigt werden. Wenn z. B. eine Instruktion auf Variablen zugreift, deren cachelines alle auf dasselbe set abgebildet werden, müssen eventuell cachelines der alten durch die der neuen Variablen ersetzt werden. Wenn die nächste Instruktion ein Datum einer verdrängten cacheline referenziert, liegt ein *conflict miss* vor. In diesem Fall hilft nur, die Adresse der Variablen so zu ändern, dass nicht mehr alle auf dasselbe set von cachelines abgebildet werden. Einige Methoden dieser sogenannten *„data layout transformations“* sollen im Folgenden vorgestellt werden.

3.4.1 Array Padding

Eine Lösung des obigen Problems besteht im *„array padding“*. Ein *„pad“* ist einfach die Deklaration einer Variablen, die zwischen die Deklaration zweier problemverursachenden Arrays eingefügt wird. Diese Variable wird nie verwendet; aber da Speicher für sie reserviert wird, bewirkt sie eine Verschiebung

der Adressen der Elemente des zweiten Arrays.

3.4.2 Array Transpose

Bei einem mehrdimensionalem Array kann die Reihenfolge der Dimensionen immer beliebig vertauscht werden. Dadurch werden auch die Adressen der innersten Arrayvariablen (bzw. ihrer cachelines) verändert. Diese Technik wird als *„array transpose“* bezeichnet.

3.4.3 Array Merging

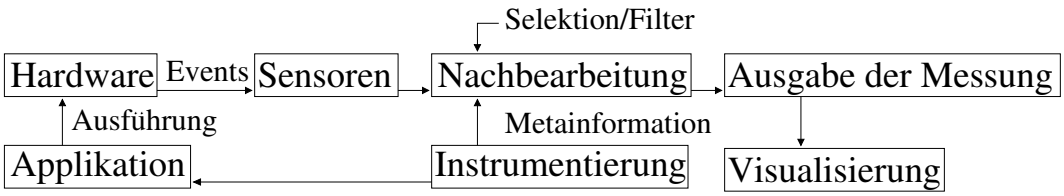
Wenn zwei Arrays denselben Typ und dieselbe Länge haben, können sie auch zu einem zweidimensionalen Array zusammengefasst werden. Es ist noch eine *array transpose* notwendig, aber das Resultat ist eine Verschiebung der Adressen der inneren Variablen des Arrays. Diese Technik nennt man *„array merging“*.

Kapitel 4

Meßwerkzeuge

Wenn ein Programmierer eine Anwendung schreibt, wird er zum Schluß versuchen, deren Performance zu steigern. Dabei wird er ihr zunächt einer groben statistischen Analyse unterziehen um festzustellen, in welchen Teilen seines Programms viel Zeit verbraucht wird und sich deshalb der Aufwand einer Optimierung lohnt. Dann wird er diesen Abschnitten einer detailierteren Untersuchung unterziehen, um möglicherweise vorhandene Schwachstellen zu finden und deren Gründe zu ermitteln. Anschließend kann er versuchen, diese Schwachstellen zu beseitigen (Z.B. mit einer der im Kapitel 3 beschriebenen Techniken, falls ein mangelhaftes Speicherzugriffsverhalten aufgedeckt wird). Bei MPI(„MessagePassingInterface“)-Applikationen kommt zu den Kosten, die die eigentliche Applikation verursucht, auch die Kosten, die die Kommunikationsroutinten hervorrufen. Im folgendem wird vorgestellt, welche Bestandteile an einer Performancemeßung beteiligt sind und anschließend Beispiele für die einzelnen Komponenten gegeben.

4.1 Allgemeine Struktur von Meßwerkzeugen



Zunächst gibt es Sensoren die die Signale der Hardware aufnehmen und weiterleiten können. Sie sorgen dafür, daß bei einer Ausführung einer Applikation die verschiedenen Events von Prozessor, Cache usw. überhaupt wahrgenommen werden können. Anschließend werden diese Events einer Nachbearbeitung unterzogen. Dabei können viele von ihnen wieder verworfen wer-

den. Eventuell muß die Applikation vor/bei der Ausführung instrumentiert werden, wenn die Nachbearbeitung gewiße Metainformationen braucht. Dann können z.B. die gesammelten Daten den Funktionen des Programms zugeordnet werden. Die von der Nachbereitungseinheit ausgegeben Meßwerte werden dann abgespeichert. Zum Schluß werden sie eventuell noch visuell dargestellt.

4.2 Simulation

Bei der Simulation wird die Funktionionalität von Hardwarekomponenten, die sich noch in der Entwicklung befinden, durch Software nachgebidet. Solche Hardwarekomponenten sind z.B. CPUs oder Hardware-Monitore. Der große Vorteil einer Simulation besteht darin, eine Hardwarekomponente testen zu koennen, ohne daß sie real exestieren muss. Wenn sich bei deren Simulation herausstellt, daß sie nicht die gewünschten Eigenschaften hat, kann sie einfach verändert werden, ohne daß große Kosten entstanden sind. Man kann sogar Tools für die projektierte Hardware entwickeln. Allerdings ist die Zeit, die eine Simulation in Anspruch nimmt, erheblich. Die Ausführung der Testprogramme wird um einen Faktor von einigen Dutzend bis einige 100 Mal verlangsamt.

4.2.1 Simulation des Caches

Den Ausführungen in Kapitel 2 zufolge, sind Werkzeuge zur Beobachtung des Cacheverhaltens eines Programms besonders interessant, da eine Optimierung des Cacheszugiffs einen hohen Performancegewinn verspricht.

cachegrind

cachegrind geht davon aus, daß die Programmausführung in U-Code –einem RISC-ähnlichen Code– simuliert wird. „Valgrind“ z.B. übersetzt die Maschinenbefehle des x86-Prozessors in U-Code. Das Speicherverhalten eines Programms kann dann leicht überwacht werden, da ja nur noch load/store-Befehle auf den Speicher zugreifen können. Infolgedessen kann cachegrind auch die Caches des Systems simuliert, die level1 instruction bzw. data caches (i.f.: I1 bzw. D1) und das gemeinsame level2 cache (i.f.: L2). Die Parameter dieser Caches auf der jeweiligen Maschine, also ihrer Größe, ihre Blockgröße und den Grad ihrer Assositivität, kann Cachegrind in der Regel automatisch ermitteln. Ansonsten geht Cachegrind von Default-Werten aus, die aber manuell überschrieben werden können. Die Cachelines werden grundsätzlich nach LRU ersetzt. Nach Programmbeendigung werden die der

I1, D1 und L2-Caches (textuell) in einer Statistik dargestellt. Vor allem wird ein Textfile angelegt, das den Zeilen des Quellprogramms die Anzahl der Misses/Hits der verschiedenen Caches zuordnet.

callgrind

Allerdings speichert cachegrind keinerlei Informationen über inividuelle Funktiosaufrufe. Deshalb kann cachegrind nur summarisch darstellen, wieviel Cachehits/-misses einer Funktion zugeordnet sind. Callgrind beseitigt diesen Mangel und speichert auch zeitliche Informationen. Die Anzahl der Cachehits/-misses kann deshalb pro Funktionaufruf angegeben werden (evt. inclusive der Kosten anderer Funktionaufrufe). Außerdem kann ein Aufrufbaum konstruiert werden.

4.3 Sensoren

4.3.1 Hardware Performance Counter

Jeder moderne Mikroprozessor verfügt über einen Satz von „Hardware Performance Counter“. Das sind Zähler, die speichern, wie oft im Prozessor ein gewißes elektronisches Signal erzeugt wird (z.B. vom Prozessor oder der Uhr), i.e. wie oft ein gewißes „Event“ auftritt. Einige Beispiele für mögliche Typen eines Events sind: Die Ausfürung eines Maschinenbefels evt., eines bestimmten Typs (integer,floating point) oder ein TLB miss. Welches Event von einem HPC beobachtet wird, kann man konfigurieren (s.u.). Realisiert sind sie in Form von prozessorinternen Registern; es gibt 2-8, abhänig vom Hersteller des Prozessors. Hardware Performance Counter können auf zweierlei Arten angewandt werden:

- Einmal kann man sie so konfigurieren, daß sie jedesmal, wenn ein bestimmts Event auftritt, eine Unterbrechung auslösen. Dadurch kann die die Häufigkeit dieses Events exakt bestimmt werden.

- Man kann sie aber auch so konfigurieren, daß sie mitzählen, wie oft ein bestimmtes Event eintritt. Sobald eine bestimmter Grenzwert überschritten wird, schicken sie eine Unterbrechung an den Prozessor. Dieser Art der Verwendung unterstützt “eventbased sampling“ (s.u.). In diesen Zusammenhang soll auf eine Besonderheit des Pentium4-Prozessors hingewiesen werden: Wenn ein Interrupt durch einen HPC auslöst wird, ist eine Kopie der Registerinhalte zum Zeitpunkts des Überlaufs verfügbar. Daraus kann auf die virtuelle Adreße geschloßen werden, die das zum

Überlauf führende Ereignis ausgelöst hat. Beim Itanium ist diese Adresse gleich in einem eigenen Register enthalten. Außerdem ist eine Filterung auf gewiße Adreß- bzw Codebereich möglich.

4.3.2 Tools zur Konfiguration von Hardware Performance Countern

perfex

„perfex“ ist ein Tool von SGI, das es ermöglicht, die HPCs von R10k und R12k-Prozessoren zu programmieren. In SGIs „Technical Publications Library“ ([5]) sind zwei Manpages, zu finden, die „perfex“ betreffen, zu finden, und zwar „PERFEX(1)“ und „PERF_COUNTERS(5)“.

perfcrt

perfcrt ermöglicht es die Hardware Performance Counter der Prozessoren bei einem Linux-Rechner zu programmieren. Folgende Prozessoren werden z.B. unterstützt: Pentium, Pentium MMX, Pentium Pro, Pentium II, Celeron, und Pentium III; AMD K7 Athlon; Cyrix 6x86MX, MII und III; und WinChip C6, 2, 2A und 3. ([4])

PAPI

PAPI („Performance Application Programming Interface“) stellt eine möglichst plattformunabhänige Schnittstelle bereit, die die „Hardware Performance Counter“ eines Prozessors zugänglich macht. Die Programmierer von „Performance-Analysern“ sollen auf diese Schnittstelle aufbauen können und so von der Aufgabe befreit werden, selbst den Zugriff auf die Counter implementieren zu müßen. Dabei sind zwei verschiedene Abstraktionsebenen vorgesehen. Zunächst kann über ein „high-level interface“ auf Counter zugegriffen werden, die in jedem Prozessor vorhanden seien sollten. Wenn nur diese kleinste gemeinsame Untermenge von Events überwacht wird, kann derselbe der Quellcode auf unterschiedlichen Plattformen ohne irgendeine Modifikation immer wieder neu kompiliert werden. Falls auf einzigartige Features des Herstellers zugegriffen werden soll, können die Counter des Prozessors über ein „low-level interface“ zugänglich gemacht werden. ([3])

4.4 Nachbearbeitung der Daten

4.4.1 Sampling

Beim „Sampling“ wird der Prozessor in gewißen Abständen unterbrochen und es werden Daten über den Zustand des Rechners gesammelt (z.B. ProcessID,ThreadID und der PC, aus dem die aktuelle Funktion des Programms hervorgeht). Diese Daten werden dann nach Beendigung des Programmlaufs statistisch ausgewertet. Man unterscheidet zwischen „timebased“ und “event-based sampling“ (TBS und EBS). Beim TBS löst ein Timerbaustein in regelmäßigen Abständen die Unterbrechungen des Prozessors aus. Beim EBS wird der Prozessor immer dann unterbrochen, wenn ein bestimmtes Event eine bestimmte Anzahl von Malen auftritt. Ein Event kann z.B. sein: eine FLOP, ein (data/instruction) cache miss, ein TLB(translation lookaside buffer) miss. EBS wird inzwischen durch „Hardware Performance Counter“ moderne Mikroprozessoren unterstützt. Man kann den Typ des Events festlegen und wie oft es auftreten muß, um eine Unterbrechung auszulösen. Der große Vorteile von Sampling besteht in dem minimalen Overhead. Man kann sich ohne große Kosten sehen, welche Teile des Programms viele CPU-Zyklen verbrauchen und genauer untersucht werden sollten.

4.4.2 Tools, die auf Sampling aufbauen

Das DCPI System

Dem DCPI („compaqs continuous profiling infrastructure“) System von Compac/HP liegt folgende Idee zugrunde: die Aktivitäten aller Executables werden unterschiedslos überwacht, also die aller Benutzerprogramme, aller shared libraries sowie die des Kernels und der Module. Daß dies mit geringem Aufwand möglich ist, liegt daran, daß das DHPI-System alle nötigen Informationen von den Hardware Performance Counter des Alpha-Prozessors erhalten kann. Der Benutzter braucht nur ein Zeitinterval, in dem Sampling stattfindet soll, festlegen. Nach dem Ende dieses Zeitinterval können verschiedene Analysetools diese Daten mehr oder weniger detailiert darstellen. ([2])

OProfile

Die Idee des DCPI-Systems von Compaq/HP –systemweites Sampling mit Hilfe von HPCs– wird von„OProfile“ übernommen und auf Linux-Systeme übertragen. In der Webpage zu „OProfile“ wird ausdrücklich darauf hingewiesen, daß die grundlegenden Prinzipien des Designs vom DCPI inspiriert

sind. Ähnlich wie bei DCPI kann man den Profiling-Prozess in zwei Phasen unterteilen:
1.Zunächst werden über jedes Stück Code, das auf dem System ausgeführt wird, Daten gesammelt. Also über Anwendungen, shared Libraries, Kernel, Module und Interrupt-handler gleichermaßen. Das Sammeln von Daten erfolgt wieder mit Hilfe von Hardware Performance Countern, weshalb der Overhead minimal ist. Während das DCPI-System jedoch auf den Alpha-Prozessoren von Compac/HP aufbaut unterstützt „OProfile“ viele verschiedene CPUs, insbesondere die von Intel, AMDs Athlon und AMD64, auch den Alpha-Prozessor und andere mehr.
2.Anschließend können diese Daten von verschiedenen Analyse-Tools ausgewertet werden.

4.5 Erzeugen von Metainformationen durch Instrumentierung

Besonderes interessant werden die Daten, die beim Sampling gesammelt werden, wenn sie gewißen Programmabschnitten zugeordnet werden können. Solche Programmabschnitte sind z.B. die Funktionen eines Programms oder auch seine Basisblöcke. Dazu müssen die Ein- und Ausstiegspunkte dieser Abschnitte irgendwann instrumentiert werden, d.h. es mußs an diesen Stellen Code eingefügt werden. Instrumentierung kann statisch oder dynamisch sein, d.h. vor der Ausführung eines Programms oder beim Laden erfolgen.

4.5.1 Meßwerkzeuge, die Instrumentierung verwenden

pixie

„pixie“ ist eine Utility von SGI für ihre Irix-Systeme. Es mißt wie oft die Basisblöcke eines Executables durchlaufen werden. Ein Basisblock ist ein maximaler Bereich von Maschinenbefehlen, der immer sequenziell ausgeführt werden. D.h. wenn ein Basisblocks ein Befehl enthält, der Ziel einer Sprunganweisung ist, muß es sein erste Befehl sein. Wenn er einen Sprungbefehl enthält, muß es sein letzter Befehl sein.
„pixie“ zerlegt nun ein binäres Programm in seine Basisblöcke, versieht sie mit zusätztlichen Code und speichert sie in einer Binärdatei. Wenn diese dann ausgeführt wird, erzeugt sie das File, daß dann von „prof“ ausgewertet werden kann. Eine manpage zu „pixie“ ist in SGIs „Technical Publications Library“ ([5]) zu finden.

vtune

Zu den komerziell verbreitesten Tools gehört „vtune“ von Intel. Zunächt unterstützt es sampling, das ausnutzen kann, sich auf die HPCs der eigenen Prozessoren stützen zu können. Außerdem wird das Programm beim Laden automatisch instrumentiert. Deshalb können die gesammelten Daten pro Modul angezeigt werden.

4.6 Ausgabe der Messungen

4.6.1 Call Graph/Call Tree

Um die Funktionen einer Anwendung überwachen zu können, müßen irgendwann ihre Ein- und Austrittspunkte instrumentiert werden. Im Falle von Call Graph werden Informationen darüber gesammelt, wie oft und vom wem die Funktion aufgerufen wurde. Aus diesen Informationen kann dann der Aufrufgraph des Programms konstruiert werden. Häufig werden auch noch die „Exclusiv-“ und „Inclusivkosten“ einer Funktion angegeben, d.h. die Summe der Kosten all ihrer Aufrufe, eclusive/inclusive der Kosten ihrer Unterprogrammaufrufe. Im Unterschied zu Call Graph speichert Call Tree zusätzlich auch noch zeitliche Informationen. Dadurch kann die Reihenfolge der Funktionsaufrufe ermittelt und in Form eines Baumdiagramms dargestellt werden. Sowohl Call Graph als auch Call Tree verursachen durch die Instrumentierung einen hohen Overhead. Bei Call Tree kommt noch dazu, daß viel Platz zur Speicherung der Daten nötig ist. Dafür kann dann der Programmfluß dargestellt werden.

4.6.2 Tracing

Beim Tracing werden die Daten, die durch EBS gewonnen werden, in einem Record gespeichert und mit einem Zeitstempel versehen. Das File, das diese Datensätze dann enthält, kann deswegen sehr umfangreich werden.
Die Auswertung dieses Tracefiles ermöglicht dann eine sehr genaue Beschreibung des Programmablaufs aber es nimmt i.a. auch sehr viel Speicherplatz in Anspruch.

4.7 Visualisierung

Sehr wichtig ist, in welcher Form am Ende des Profiling die Daten dargestellt werden. Z.B. können Informationen über die Aufteilung der Ausführungszeit

eines Programms auf die verschiedenen Funktionen als textbasierte Tabelle mit prozenturalen Angaben oder als Histogramm dargestellt werden. Letzterer Form sind die gewünschten Informationen sicherlich einfacher zu entnehmen. Deshalb gibt es verschiedene Tools, die den Zweck haben, die Daten, die andere Tool gesammelt haben, graphisch darzustellen.

4.7.1 Beispiele für Visualisierungwerkzeuge

kcachegrind

„kcachegrind“ kann die (zunächst textbasierte) Ausgabe von cache-/callgrind in visueller Form darstellen. Es kann aber auch der Output anderer Profiler visualisiert werden z.B. von Oprofile, wozu nur ein Konvertierungsskript nötig ist.

Vampir

Die offensichtlichste Anwendung von „Vampir“ ist, die Daten des Tracefiles, das „Vampiretrace“ angelegt hat, graphisch anzuzeigen. Zunächst können verschiedene Fenster mit statistischen Informationen geöffnet werden. Das wesentliche Feature von „Vampir“ sind aber die „timelines“. Sie stellen dar, wie die Aktivität des Prozessors zwischen Applikation und dem MPI bzw. den Prozessen der Applikation und den zugehörigen MPI-Routinen im Abhänigkeit von der Zeit aufgeteilt sind. Wenn man eine detailierte Darstellung will, kann man sich einfach per Mausklick auf einen schmaleren Abschnitt der Zeitachse einzoomen.

4.8 Spezialitäten der Cachemessung

Hier sind mögliche Events: Die Anzahl der Fehlzugriffe auf die Daten- bzw. Instruktions-Caches (instruction/data miss) in verschiedenen Ebenen und wie oft ein Zeile eines Caches durch einen Schreibzugriff ungültig gemacht wird (invalidation).

4.9 Mängel gegenwärtiger Performancetools

1. Ein großer Nachteil gegenwärtigen Tools ist, daß i.a. ein Ereignis eines bestimmten Typs immer gleich behandelt wird, unabhänig davon, von welcher Variablen es ausgelöst wird. Man kann dem Ereignis nur einem Bereich von Programmzeilen zuordnen, aber keiner Speicheradresse.

Wenn insbesondere ein Peak bei den cache misses festgestellt wird, weiß man die auslösenden Variablen nicht. Es wäre sehr wünschenswert, wenn Tools zur Performance Analyse einem Event die Variablen zuordnen würde, die es ausgelöst hat. Es gibt zwei Prozessoren, die eine solche Zuordnung ermöglichen, und zwar der Pentium 4 und der Itanium.

2. Selbst wenn der Programmierer aus irgendwelchen Gründen wissen sollte welche Variablen für die Optimierung relevant sind (z.B. indem er aus der Zuordnung der Events auf die Zeilen des Quellcodes Schlüße zieht), könnte er das Meßwerkzeug nicht anweisen, nur diese Variablen, zu überwachen. Er muß immer die Kosten tragen, die *alle* Variablen verursachen; diese können dadurch sehr groß werden. In diesem Zusammenhang sei auf das EP-Cache-Projekt verwiesen, das genau dieses Ziel hat, nämlich einen Hardwaremonitor zu entwerfen, der die Beobachtung einiger ausgewählter Variablen gestattet [1].

Die „data_mapping_lib“ soll dazu beitragen, genau diese Mängel zu beheben, wie im folgendem Kapitel genauer erklärt wird.

Kapitel 5

Aufgabe und Lösungskonzept

5.1 Probleme bei der Performance-Analyse

In Kapitel 4 hat man gesehen, wie man Performance-Mängel eines Executables aufdecken kann. Dazu gehören insbesondere Bereiche eines Programms, bei denen die Datenzugriffe zu vielen „cache misses“ führen und deshalb viele CPU-Zyklen verbrauchen. Allerdings sind auch zwei Probleme gegenwärtiger Performance Measurment Tools deutlich geworden:

1. Um das Cacheverhalten in einem Abschnitt eines Programms analysieren und verbessern zu können, müssen diejenigen Variablen, die die cache misses in diesem Abschnitt ausgelösen, bekannt sein. Genau zwei Prozessoren (der Pentium 4 und der Itanium) enthalten (direkt oder indirekt; evt. ist eine Berechnung nötig) bei einem cache miss die Adresse der Variablen, die es ausgelöst hat. Aber selbst hier liegt die Information nur in einer wenig brauchbaren Form, nämlich in Form von einer Speicheradresse vor. Der Programmierer benötigt aber die Namen der Variablen, die die cache misses verursachen, um den Sourcecode seines Programms verbessern zu können (z.B. durch eine Kapitel 3 vorgestellten Techniken).

2. Durch die Performanceanalyse entstehen immer mehr oder weniger hohe Kosten hinsichtlich Speicherplatz- und Zeitverbrauch. Beim Sampling sind sie noch minimal, aber wenn man detailierte Informationen über den Programmblauf erhalten will, sind die entstehenden Kosten nicht mehr vernachläßigbar gering. Eine selektive Auswahl der zu beobachtenden Variablen ist notwendig, um die Kosten zu verhindern, die mit sehr großen Datenmengen vebunden sind. Bei der Simulation eines Hardwaremonitors z.B. ist die Datenmenge immens und infolgedessen

auch die entstehenden Kosten, wenn beobachteten Variablen nicht eingeschränkt werden.

5.2 Aufgabenstellung

Die Aufgabe dieser Diplomarbeit war es, ein Bibiothek zu erstellen, die die obengenannten Probleme zu lösen hilft. Die dabei entstande Bibliothek kann eine Speicheradresse in den (eindeutig qualifizierten) Namen einer Variablen abbilden, sowie ihrer Grösse und ihren Typ. Umgekehrt kann sie den Namen einer (zum Zeitpunkt der Programmausführung lebenden) Variablen auf ihren Adressbereich abbilden. Folgende Einschränkungen sind zu beachten:

1. Die Bibliothek wurde auf einem Linux-System mit x86-Prozessor entwickelt und nie woanders getestet.

2. Sie setzt DWARF2 als Debuggingformat voraus.

3. Es werden nur statische/globale sowie Variablen unterstützt; keine dynamischen Bereiche.

5.3 Lösungen mit Hilfe der „data_mapping_lib“

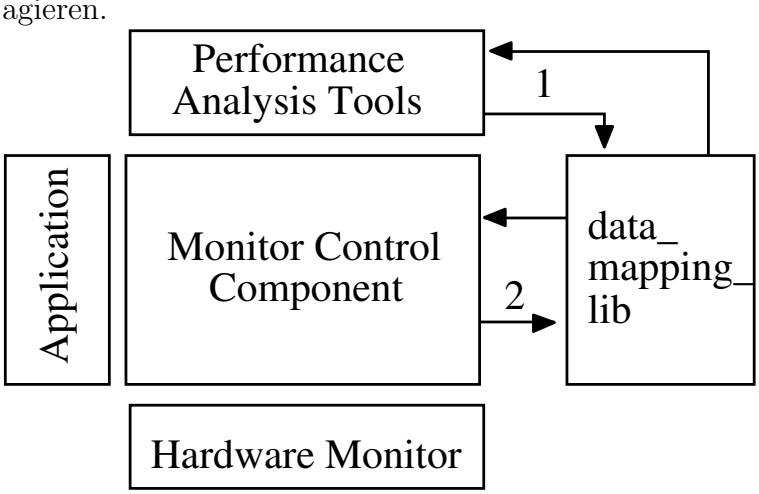
Das erste Probleme lässt dadurch lösen, dass das Performance Measurement Tool ein (Cache-)Event zusammen mit der Adresse, die es ausgelöst hat, erhält. Außerdem muss es auf eine Funktion zugreifen können, die eine Adresse in den Namen der Variablen übersetzen kann, die zum gegenwärtigen Zeitpunkt der Programmausführung unter dieser Adresse gespeichert wird. Diese Funktion muss auch Daten zurückgegeben, die es ermöglichen, den Namen eindeutig zu identifizieren, weil ja z. B. verschiedene Unterprogramme denselben Namen für eine lokale Variable verwenden können.

Durch den Einsatz eines Hardware-Monitors, der das Filtern nach Adressbereichen ermöglicht, kann auch das zweite Problem gelöst werden. Ausserdem muss es eine Kontroll-Komponente geben, die diesen konfigurieren, und auch auf eine „Umkehrfunktion“ zugreifen kann. Unter einer „Umkehrfunktion“ ist eine Abbildung vom (eindeutig qualifizierten) Variablennamen auf *alle* Adressen, die zu zu einem bestimmten Zeitpunkt der Programmausführung zu dieser Variablen gehören, zu verstehen. Auch eine solche Abbildung enthält die „data_mapping_lib“; sie baut natürlich auf deselben internen Liste auf, wie die andere Lookup-Funktion. Das Performance Measurement Tool kann

dann der Kontroll-Komponent in Form von Namen miteilen, welch Variablen es überwachen will. Die Kontroll-Komponente ruft dann die Lookup-funktion auf, um die Variabelennamen in Adressen zu übersetzten, und konfiguriert dann den Hardwaremonitor entsprechend. Dann werden nur noch einige wenige ausgewählte Variablen beobachtet werden und geringere Kosten entstehen.

5.4 Anwendung im Meßwerkzeug

Wie man in Abschnitt 5.3 gesehen hat, könnte ein Hardwaremonitor in Verbindung mit der data_mapping_lib beide obengenannten Probleme lösen. Es sei hier also nochmal dargestellt, wie die verschiedenen Komponenten interagieren.



Abbidung 5.1 .
Die zentrale Rolle spielt die „Monitor Control Component“ (MCC). Allgemein gesagt regelt sie die Kommunikation zwischen dem Hardwaremonitor und dem Performance Measurement Tool und kann z. B. den Namen einer Variablen in deren Adresse übersetzten. Dazu muss die MCC auf die Dienste der „data_mapping_lib“ zugreifen können. Insbesondere kann MCC kann den Hardwaremonitor nach den Vorgaben des Performance Measurement Tool konfigurieren. In Abbildung 5.1 sind zwei Fälle zu unterscheiden:

1. Das Performance Measurement Tool erhält ein (Cache-)Event zusammen mit der Adresse, die es ausgelöst hat. Anschließend ruft es eine Funktion auf, die die Adresse in den Namen der Variablen übersetzt (mit „1“ markiert).

2. Das Performance Measurement Tool will nur ausgewählte Variablen beobachten und teilt deren (qualifizierten) Namen dem MCC mit. Diese

ruft eine Funktion auf, die die Namen der Variablen in deren Adressen übersetzt und konfiguriert den Hardwaremonitor entsprechend (mit „2“ markiert).

Insgesamt kann das Performance Measurement Tool die data_mapping_lib also in folgenden Schritten angewenden:

1. In einer ersten globalen Meßung werden die Datenstrukturen ausfindig gemacht, die ein viele cache misses verursachen und deshalb optimiert werden sollten.

2. Bei der anschließenden Meßung beschränkt man sich auf diese Datenstrukturen und spart dadurch Zeit und Speicherplatz.

5.5 Realisierung der „data_mapping_lib“

Beide Abbildungen (die zwischen einer Hauptspeicheradresse eines Datums und den Namen der zugehörigen Variablen bzw. die vom Namen einer Variablen auf alle zugehörigen Speicheradressen) bauen auf derselben Grundlage auf: Bei der Compilierung des Quellcodes der Applikation müssen Debuginformationen erzeugt werden. Beim Start des Executables werden diese dann ausgewertet. Variable können in drei Klassen unterteilt werden: statische, Stack- und dynamische Variablen. Jede Klasse macht eine unterschiedliche Vorgehensweise erforderlich.

1. Der einfachste Fall sind die statischen oder globalen Variablen, also die Variablen, die außerhalb einer Funktion deklariert werden. Ihre Adressen stehen schon nach der Kompilierung des Programms fest und können bei dessen Start direkt den Debug-Informationen entnommen werden.

2. Bei den Stackvariablen (also den formalen Parametern und lokalen Variablen eines Unterprogramms) ist es etwas komplizierter. Ihre Adresse steht erst zur Laufzeit fest und muss als Summe der Framebase der zugehörigen Funktion und des Offsets der Variablen zur Framebase berechnet werden. Immerhin ist der Debuginformation der der zweite Summand zu entnehmen.

3. Der letzte Fall sind die dynamischen Variablen. Da es hier natürlich höchstens Debuginformationen über die zugehörigen Pointer geben kann, muss die data_mapping_lib die meiste Arbeit übernehmen. Immer wenn “malloc()“ (oder „calloc()“ oder „realloc()“ oder „free()“) aufgerufen

wird, muss der Speicherbereich, der alloziert wurde, ermittelt und einer dereferenzierten Zeigervariablen zugewiesen (bzw. entzogen) werden.

Im DWARF-Format, auf dem die „data_mapping_lib“ aufbaut, haben diese Debuginformationen folgende Form.

- Bei einer statisch/globale Variable wird das „location“-Attribut in Form einer absoluten Adresse spezifiziert:

  ```
  DW_OP_addr <0x...>
  ```

- Die Adresse eines formalen Parameters oder einer lokalen Variablen wird folgendermaßen spezifieziert:

  ```
  DW_OP_fbreg <offset>
  ```

  Das fb-Register enthält aber nicht die Adresse der Basis des Stackframes, insofern bei DWARF die formalen Parameter einen positiven und die lokalen Variablen einen negativen Offset haben.
  Die Adresse der (verschobenen) Framebase wird dann von instrumentiertem Code zur Laufzeit des Executables ermittelt. Dazu müssen an den Ein- und Austrittspunkten der überwachten Unterprogramme Funktionensaufrufe in den Quellcode des Programms eingefügt werden. Dann können die Adressen der berechnet und in die list_of_vars eingefügt werden. Man beachte, dass die formalen Parametern und lokalen Variablen aller Funktionen so berechnet werden müssen, also insbesondere auch die Stackvariablen von main(), obwohl bei ihnen die Adresse schon zum Zeitpunkt der Kompilierung feststeht.

Die dynamischen Variablen sind in der data_mapping_lib noch nicht implementiert. Wenn die dynamischen Variablen auch in die list_of_vars aufgenommen werden, ist unbedingt darauf zu achten, dass sie am hinteren Ende der Liste angehängt werden. Das vordere Ende ist für die Stack-Variablen vorgesehen. Die dmlib_exit() baut auf dieser Anordnung auf und entfernt einfach die Anzahl der Variablen, die dem Unterprogramm zugeordnet ist, vom Stack-Ende der list_of_vars.

Kapitel 6

Aufbau und Funktionalität der Schnittstelle

6.1 Das Anwenden der data_mapping _lib

Im Folgenden wird der Zyklus beschrieben, der durchlaufen wird, wenn das Speicherzugriffsverhalten einer Anwendung verbessert werden soll.

- Bevor das MCC auf die Funktionalität der data_mapping_lib zugreifen kann, sind folgende vorbereitende Schritte auszuführen:
  - Zunächst einmal muss das Programm, dessen Speicherzugriffsverhalten überwacht werden soll, instrumentiert werden. Dazu muss es in der Form von Quellcode vorliegen. Der Code, der dabei eingefügt wird, includiert zunächst die „data_mapping_lib“. Anschließend werden einige ihrer Funktionen aufgerufen (s. u.). Diese Aufrufe sorgen dafür, dass zur Laufzeit des instrumentierten Programms eine Liste aller lebenden Variablen aufgebaut und immer wieder aktualisiert wird.
  - Um die list_of_vars initialisieren und updaten zu können, muss die data_mapping_lib beim Start des instrumentierten Executables die die DWARF-Debuginformationen auswerten. Diese müssen beim Kompilieren folglich erzeugt werden. Die „data_mapping_lib“ baut auf der „libdwarf“ [6] auf und diese verwendet wiederum die „libelf“ [7]. Deshalb müssen neben der data_mapping_lib.a auch die libdwarf.a und die libelf.so zum Programm gelinkt weden.
- Die MCC kann nun auf die Lookup-funktionen der zugreifen und bei einem cache event die Adresse der zugehörigen Variablen in den Namen

der Variablen übersetzten, bevor sie es an das Performance Measurement Tool weitereicht. Bei einer ersten Ausführung des Executable welche Variablen aufgedeckt, die viele cache misses verursachen und deshalb Kanidaten für eine Optimierung sind.

- Jetzt sind die Namen der Variablen bekannt, die durch viele Cachefehlzugriffe eine schlechte Performance verursachen. Der Programmierer kann dem Performance Measurement Tool anweisen bei einer zweiten Ausführung des Executable genauere Informationen über die betreffenden Variablen zu sammeln.

- Die gesammeten Daten sollten jetzt Auskunft darüber geben, wie das schlechte Cacheverhalten zu erklären ist. Der Programmierer kann sein Programm entsprechend abändern.

- Nach einem erneuten Kompilieren kann er die Performance seines Programms testen. Falls er sie zufriedenstellend findet, ist er fertig. Falls nicht, muss er diesen Zyklus wiederholen.

6.2 Bei der Instrumentierung eingefügte Funktionen

6.2.1 Die Initialisierung

Zu Beginn der Ausführung des instrumentierten Programms müssen erstmal die globalen Datenstrukturen der „data_mapping_lib“ initialisiert werden, also die „list_of_vars“, die „table_of_subprogramms“ und die „number_of_subprogramms“. Dies geschieht durch einen Aufruf von „dmlib_init_lib(..)“ gleich zu Beginn der main()-Funktion des instrumentierten Programms. Sie ermöglichen es das instrumentierte Executable zu öffnen und die Debugginformationen der verschiedenen „.debug_*“-Sektionen zugänglich zu machen. Insbesondere kann jetzt auf die „.debug_info“-Sektion, die die „Debugging Information Entries“ enthält, zugegiffen werden. Nun können die globalen Strukturen initialisiert werden. insbesondere die „list_of_vars“. Sie ist der wesentliche Bestandteil der „data_mapping_lib“. In ihr sind alle zu einem bestimmten Zeitpunkt lebenden Variablen gespeichert. Hier wird sie erstmal mit den statischen Variablen initialisiert und muss später immer wieder aktualisiert werden (ref). Außerdem werden die „number_of_subprogramms“ und die „table_of_subprogramms“ angelegt. Sie enthalten Informationen, die später bei einem Unterprogrammaufruf benötigt werden, um dessen formale Parameter und lokale Variablen zur „list_of_vars“ hinzuzufügen bzw. zu entfernen.

```
int dmlib_init_lib(int argc, char** argv)
```

FORMALE PARAMETER:
argc, argv:
Diese Parameter werden von der MCC der main()-Funktion des zu messenden Programms entnommen und an die dmlib_init_lib() weitergereicht.

RÜCKGABEWERTE:
1:
Wenn die zu messende Applikation mit der Option „–dmlib_print_CUs“ wurde und die Compile Units erfolgreich ausgedruckt werden konnten, wird 1 zurückgegeben.
-1:
Bei einem schwerwiegenden Fehler, der im übergeordneten Programm (bei der MCC) zum Programmabruch führen sollte, wird -1 zurückgegeben. Schwerwiegende Fehler werden der können nur in den aufgerufenen Unterprogrammen auftreten und werden der durch einen negativen Rückgabewert mitgeteilt.
0:
Ansonsten wird 0 zurüchgegeben.

6.2.2 Der Aufruf eines Unterprogramms

Bei der Instrumentierung muss außerdem in der Definition eines Unterprogramms vor der ersten Anweisung die Funktion „dmlib_entered.c()“ aufgerufen werden. Dieser Aufruf sorgt dafür, dass die formale Parameter und lokale Variablen des Unterprogramms in die „list_of_vars“ aufgenommen werden.

```
int dmlib_entered(int file_id,int line_nr,\
          char* subprogram_name,char* dummy_address);
```

file_id, line_nr: Werden später von den Lookup-funktionen zu einer eindeutigen Identifizierung eines formalen Parameter einer lokalen Variablen des Unterprogramms benötigt.

subprogram_name: Der Name des aufgerufenen Unterprogramms. Er ist ntig, um den richtigen Eintrag in der 'table_of_subprograms' zu finden.

dummy_address: Bei der Instrumentierung muss in jedem Unterprogramm eine Variable vom Typ char angelegt werden, deren Name mit der Zeichenkette 'dummy' beginnt. Die Adresse dieser Dummyvariablen muss der dmlib_entered() als Parameter uebergeben werden. Zusammen mit

ihrem Offset, der in der 'table_of_subprogramms' gespeichert ist, kann die Basisadresse des Stacks berechnet werden.

Rückgabewerte:

-1 Bei einem schwerwiegenden Fehler, der im übergeordneten Programm (bei der MCC) zum Programmabruch führen sollte, wird -1 zurückgegeben. Schwerwiegende Fehler sind:
- table_of_subprograms==NULL
- das Unterprogramm kann in der table_of_subprograms nicht gefunden werden

0 Im Erfolgfall wird 0 zurückgegeben.

6.2.3 Die Beendigung eines Unterprogramms

Umgekehrt muss vor der letzten Anweisung eines Unterprogramms die Funktion „dmlib_exit.c()“ aufgerufen werden. Diese Funktion sorgt dafür, dass die formalen Parameter und lokalen Variablen des Unterprogramms wieder von der list_of_vars entfernt werden.

```
int dmlib_exit (char* subprogram_name);
```

subprogram_name: Der Name des aufgerufenen Unterprogramms. Er ist ntig, um den richtigen Eintrag in der 'table_of_subprograms' zu finden.

Rückgabewerte:

-1 Bei einem schwerwiegenden Fehler, der im übergeordneten Programm (bei der MCC) zum Programmabruch führen sollte, wird -1 zurückgegeben. Schwerwiegende Fehler sind:
- table_of_subprograms==NULL
- das Unterprogramm kann in der table_of_subprograms nicht gefunden werden

0 Im Erfolgfall wird 0 zurückgegeben.

Die Parameter file_id und line_nr werden diesmal nicht gebraucht, weil ja der Name des Unterprogramms bekannt ist und deshalb auch die Anzahl seiner formalen Parameter und lokalen Variablen. Diese können dann einfach vom „Stackende“ der list_of_vars entfernt werden.

6.2.4 Die Allozierung von Speicher

Immer wenn dynamischer Speicherplatz angefordert wird, kann die *dmlib_malloc()* aufgrufen werden.

```
int dmlib_malloc(long unsigned start,long unsigned end,\
                 int file_id,int line_nr,char* name);
```

start,end: Der Adressbereich, der alloziert werden soll.

file_id, line_nr: Werden später von den Lookup-funktionen und der dmlib_free() zu einer eindeutigen Identifizierung des Pointers benötigt.

name: Der Name des Pointers, fr den der Speicher reserviert wird.

6.2.5 Die Rückgabe von Speicher

Wenn der Speicher wieder freigegeben werden soll, muss die *dmlib_free()* aufgrufen werden.

```
int dmlib_free(int file_id,int line_nr,char* name);
```

file_id, line_nr: Sie ermöglichen eine eindeutigen Identifizierung des Pointers, falls er denselben Namen hat, wie eine andere Variable.

name: Der Name des Pointers, dessen Speicher zurückgegeben wird.

6.3 Lookupfunktionen

6.3.1 Die Abbildung einer Adresse einer Variablen auf ihren Namen

```
int dmlib_get_name(long unsigned address,\
              int* file_id,int* line_nr,char** name);
```

address: Eine beliebige Adresse, die zu der gesuchten Variablen gehrt.

file_id,line_nr: Es werden die Werte aus dmlib_entered() des jeweiligen Unterprogramms ausgegeben. Sie ermöglichen eine eindeutigen Identifizierung der Variablen, falls sie denselben Namen hat, wie eine andere Variable. Wenn beide Parameter den Wert 0 haben, ist die gesuchte Variable eine statische Variable.

name: Der Name der gesuchten Variablen (output), falls er in der list_of_vars gefunden werden konnte.

Rückgabewerte:

-1 Signalisiert der MCC, dass kein Name zu der Adresse in der list_of_vars zu finden ist.

0 Bei erfolgreicher Suche wird 0 zurückgegeben.

6.3.2 Die Abbildung eines Variablennamens auf alle zugehörigen Adressen

Wenn mit dmlib_get_name() bestimmt wurde, welcher Variablenname zu einer Adresse gehört, können mit dmlib_get_range() auch die übrigen Adressen, die zu dieser Variablen gehören, ermittelt werden.

```
int dmlib_get_range(char* name,int file_id,int line_nr,\
                    unsigned long* start,unsigned long* end);
```

name: Der Name der Variablen, deren Adressbereich gesucht wird.

file_id,line_nr: Es sind die Werte aus dmlib_entered() des jeweiligen Unterprogramms anzugeben. Sie ermöglichen eine eindeutigen Identifizierung der Variablen, falls sie denselben Namen hat, wie eine andere Variable. Wenn beide Parameter den Wert 0 haben, wird nach statischen Variablen gesucht.

name: Der Name der gesuchten Variablen (output), falls er in der list_of_vars gefunden werden konnte.

Rückgabewerte:

-1 Signalisiert der MCC, dass kein Name der Variablen in der list_of_vars zu finden ist.

0 Bei erfolgreicher Suche wird 0 zurückgegeben.

file_id,line_nr: wie oben.

Kapitel 7

Der interne Aufbau der „data_mapping_lib“

7.1 Die globalen Strukturen

7.1.1 Die „list_of_vars“

Wie schon mehrfach gesagt, ist die wesentliche Struktur der „data_mapping_lib“ die „list_of_vars“. Sie ist ein Pointer auf dem Typ „var_t“, der folgendermaßen definiert ist:

```
typedef struct{
      char*                   var_name;
      long unsigned           startaddress;
      long unsigned           endaddress;
      int                     line_nr;
      int                     file_id;
      struct varlistelement   *next;
}var_t;
```

var_name: Der Name der Variablen.

startaddress: Der Startadresse der Variablen.

endaddress: Der Endadresse der Variablen.

line_nr,file_id: Diese Parameter dienen einer eindeutigen Identifizierung der Variable bei Namensgleichheit. Beide haben bei statischen Variablen den Wert 0; formalen Parameter und lokalen Variable erhalten den Wert, der ihrem Unterprogramm bei der Instrumentierung zugeteilt wird.

next: Ein Zeiger auf das nächste Element der list_of_vars.

7.1.2 „number_of_subprograms“

Die Integer-Variable number_of_subprograms wird dazu benötigt, den Speicherplatz zu berechnen, der für die „table_of_subprograms“ angelegt werden muss.

7.1.3 Die „table_of_subprograms“

Jedem Unterprogramm des überwachten Executables ist ein Eintrag in der table_of_subprograms zugeordnet. Er ist folgendermaßen aufgebaut:

```
typedef struct subprogram{
      char*                       subprogram_name;
      int                         number_of_form_param_and_loc_vars;
      form_param_and_loc_var_t*   table_of_form_param_and_loc_vars;
}subprogram_t;
```

subprogram_name: Der Name des jeweiligen Unterprogramms.

number_of_form_param_and_loc_vars: Die Anzahl seiner formalen Parameter und lokalen Variablen.

table_of_form_param_and_loc_vars: Ein Zeiger auf eine Tabelle aller formalen Parameter und lokalen Variablen des Unterprogramms.

Die Einträge in diese Tabelle beschreiben jeweils einen formalen Parameter oder eine lokale Variable und haben folgenden Aufbau:

```
typedef struct form_param_or_loc_var{
      char*            name;
      int              offset;
      int              byte_size;
}form_param_and_loc_var_t;
```

name:

offset:

byte_size:

7.2 Initialisierung

7.2.1 Die Initialisierung der „data_mapping_lib“

Bevor die Update- und Lookupfunktionen auf die „list_of_vars“ zugreifen können, müssen zu Beginn der Ausführung der überwachten Applikation erstmal die globalen Datenstrukturen der „data_mapping_lib“ initialisiert werden. Das sind die „list_of_vars“, die „number_of_subprograms“, die „table_of_subprograms“ und damit indirekt für jeden Eintrag in die „table_of_subprograms“ eine „table_of_form_param_and_loc_vars“. Die „list_of_vars“ wird zur Laufzeit immer wieder modifiziert; die anderen globalen Strukturen bleiben im Folgenden unverändert und werden im wesentlichen dazu gebraucht um, diese Modifikationen durchzuführen. Dazu wird am Anfang der „main()“-Funktion der überwachten Applikation die „dmlib_init_lib.c()“ aufgerufen. Alle internen Funktionen werden direkt oder indirekt von ihr aufgerufen. Die folgende Abbildung soll einen Überblick geben; danach werden die beteiligten Funktionen genauer beschrieben.

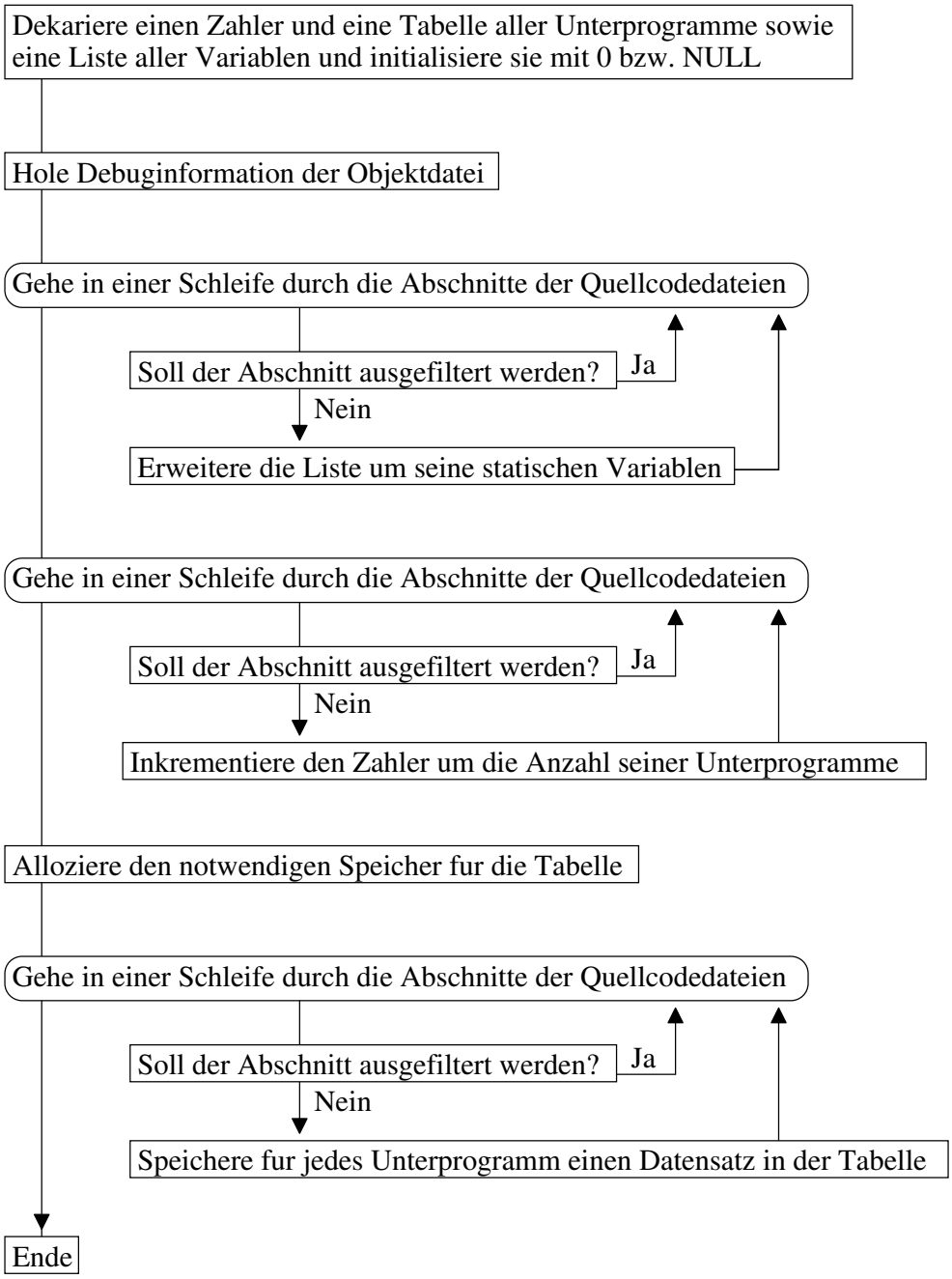


Abbildung 7.1 In der ersten while()-Schleife wird die „list_of_vars“ mit den statischen Variablen initialisiert. In der zweiten while()-Schleife werden die Unterprogramme des Executables abgezählt. Danach kann der nötige Speicher für die „table_of_subprograms“ alloziert werden. In der dritten while()-Schleife wird die „table_of_subprograms“ dann ausgefüllt.

7.2.2 Die Initialisierungsfunktionen

Die „dmlib_init_file()“

```
int dmlib_init_file(int argc, char** argv, Dwarf_Debug* dbg)
```

Die Aufgabe der dmlib_init_file() besteht darin, einem Handle, das die Debuginformationen der überwachten Applikation zugänglich macht, einen Wert zuzuweisen. Im Folgenden kann es von anderen Funktionen benutzt werden.

formale Parameter:

argc, argv Diese Parameter werden von der MCC der main()-Funktion der überwachten Applikation entnommen und an die dmlib_init_lib() weitergereicht, die sie wiederum der dmlib_init_file() übergibt.

dbg Ein Handle, das die Debuginformationen des Executables zugänglich macht.

Rückgabewerte:

1 Signalisiert der „dmlib_init_lib.c()“ „dmlib_print_compile_units()“ aufzurufen und 1 zurückzugeben, um der MCC einen Programmabbruch nahezulegen, nachdem die dmlib_init_file() eine Liste aller CUs erfolgreich ausdrucken konnte.

0 Das übergebene File (arg[0]) konnte geöffnet und initialisiert (mit dwarf_init()) werden.

-1 Das übergebene File (arg[0]) konnte nicht geöffnet oder nicht initialisiert (mit dwarf_init()) werden. Die „dmlib_init_lib.c()“ sollte daraufhin -1 zurückzugeben, um der MCC einen Programmabbruch nahezulegen.

Die „dmlib_get_root_DIE_of_next_CU()“

```
int dmlib_get_root_DIE_of_next_CU\
     (Dwarf_Debug dbg,\
      Dwarf_Die* root_die){
```

Die dmlib_get_root_DIE_of_next_CU() hat die Aufgabe bei jedem Aufruf zur nächste CU der überwachten Applikation auszuwählen und einen Zeiger auf die Wurzel des zugehörigen DIE-Baums zurückzugeben. Wenn alle CUs abgearbeitet sind, wird das dem aufrufendem Proramm mitgeteilt (s.u.).

formale Parameter:

dbg Ein Handle, das die Debuginformationen des Executables zugänglich macht.

root_die Ein Zeiger auf die Wurzel des DIE-Baums der aktuellen CU. Die Aufgabe der dmlib_get_root_DIE_of_next_CU() besteht darin, diesem Zeiger einen Wert zuzuweisen, so dass er von der folgenden Funktion benutzt werden kann.

Rückgabewerte:

2 Die CU ist in der dmlib_filter.conf enthalten und sollte deshalb ausgefiltert werden.

1 Signalisiert dem aufrufenden Programm, dass die letzte CU schon ausgegeben wurde und deshalb eine Schleife, in der alle CUs durchlaufen werden, verlassen werden kann.

0 Die CU wurde dem Parameter root_die erfolreich zugewiesen.

-1 Der Aufruf von dwarf_next_cu_header() hat zu einem Fehler geführt. Die „dmlib_init_lib.c()“ sollte daraufhin -1 zurückzugeben, um der MCC einen Programmabbruch nahezulegen.

-2 Die die CU konnte zwar gefunden werden; aber nicht die Wurzel des zugehörigen DIE-Baums. Die „dmlib_init_lib.c()“ sollte daraufhin -1 zurückzugeben, um der MCC einen Programmabbruch nahezulegen.

-4 Sowohl die CU als die Wurzel des zugehörigen DIE-Baums konnten gefunden werden; aber nicht dessen Name. Die „dmlib_init_lib.c()“ sollte daraufhin -1 zurückzugeben, um der MCC einen Programmabbruch nahezulegen.

Im Rumpf der while()-Schleifen wird die „dmlib_get_root_DIE_of_next_CU()“ aufgerufen. Sie springt von der ersten bis zur letzten „Compile Unit“ und stellt den formalen Parameter „root_DIE“ auf die Wurzel des zugehörigen DIE-Baums. Wenn alle „Compile Units“ abgearbeitet sind, wird das von der „dmlib_get_root_DIE_of_next_CU()“ signalisiert, indem sie „1“ statt „0“ zurückgibt. Dies wird von der „dmlib_init_lib()“ dazu benutzt, eine while()-Schleife zu verlaßen und die nächste zu betreten. Beim nächsten Aufruf gibt die „dmlib_get_root_DIE_of_next_CU()“ dann wieder den ersten Wurzelknoten der ersten CU des Executables aus.

Die „dmlib_get_static_vars()“

```
int dmlib_get_static_vars(Dwarf_Debug dbg,Dwarf_Die root_die,\
                          var_t** ptr_to_ptr_to_list_of_vars
```

formale Parameter:

dbg Ein Handle, das die Debuginformationen des Executables zugänglich macht.

root_die Ein Zeiger auf die Wurzel des DIE-Baums der aktuellen CU.

ptr_to_ptr_to_list_of_vars Eine Zeiger auf einen Zeiger auf den Anfang der list_of_vars

Rückgabewerte:

0 Im Erfolgsfall wird 0 ausgegeben.

-1 Fehler, der zu Programmabruch fhrt sollte.

-2 Fehler, bei dem nur eine Warning ausgegeben wird.

Die „dmlib_count_subprograms()“

```
int dmlib_count_subprograms(Dwarf_Debug dbg, \
          Dwarf_Die root_die,int* ptr_to_number_of_subprograms)
Die Aufgabe der dmlib\_count\_subprograms()
besteht darin, die Unterprogramme der
"uberwachten Applikation abzuz"ahlen.
Im folgendem kann die dmlib\_init\_lib()
dann den n"otigen Speicherplatz f"ur
die table\_of\_subprograms allozieren.
```

formale Parameter:

dbg Ein Handle, das die Debuginformationen des Executables zugänglich macht.

root_die Ein Zeiger auf die Wurzel des DIE-Baums der aktuellen CU.

ptr_to_number_of_subprograms) Ein Zeiger auf die globale Variable number_of_subprograms, die nach Ende aller Aufrufe der dmlib_count_subprograms() die Anzahl Unterprogramme enthält.

Rückgabewerte:

0 Im Erfolgsfall wird 0 ausgegeben.

-1 Fehler, der zu Programmabruch fhrt sollte.

-2 Fehler, bei dem nur eine Warning ausgegeben wird.

„dmlib_fill_out_table_of_subprograms()“

```
int dmlib_fill_out_table_of_subprograms(Dwarf_Debug dbg, \
          Dwarf_Die root_die, \
          subprogram_t* table_of_subprograms, \
          int* ptr_to_index_of_table_of_subprograms){
```

Die Aufgabe der dmlib_fill_out_table_of_subprograms() besteht darin, für jedes Unterprogramm der überwachten Applikation ein Record vom Typ subprogram_t zu erstellen und in der table_of_subprograms abzuspeichern. Diese Information wird später benötigt, um die list_of_vars bei einem Unterprogrammaufruf aktualisieren zu können.

formale Parameter:

dbg Ein Handle, das die Debuginformationen des Executables zugänglich macht.

root_die Ein Zeiger auf die Wurzel des DIE-Baums der aktuellen CU.

table_of_subprograms Ein Zeiger auf die globale Variable table_of_subprograms.

ptr_to_index_of_table_of_subprograms:

Rückgabewerte:

0 Im Erfolgsfall wird 0 ausgegeben.

-1 Fehler, der zu Programmabruch fhrt sollte.

-2 Fehler, bei dem nur eine Warning ausgegeben wird.

Die „dmlib_filter_compile_unit()“

```
int dmlib_filter_compile_unit(char *compile_unit_name);
```

formale Parameter:

compile_unit_name Der Name der CU.

Rückgabewerte:

1 Name der CU wurde in dmlib_filter.config gefunden und sollte augefiltert zu werden.

-1 Die dmlib_filter.conf konnte nicht geöffnet werden z.B. weil sie nicht exestiert. Das aufrufende Programm sollte diesen Fehler also igonorieren.

0 Name der CU wurde in dmlib_filter.config nicht gefunden und braucht nicht augefiltert zu werden.

Falls die „dmlib_get_root_DIE_of_next_CU()“ den Wert „0“ zurückgibt, wird neue CU einer Filterung unterzogen. Dazu wird die „dmlib_filter_compile_unit()“ aufgerufen. Sie sieht in der „dmlib_filter.conf“ (siehe 8.3) nach, ob der Name der „Compile Unit“ hier enthalten ist. Wenn ja, wird die CU ausgefiltert (also zum Beginn der while()-Schleife zurückgesprungen); andernfalls wird sie an eine der drei eigentlichen Initialisierungsfunktionen weitergereicht.

7.3 Die Konfigurationsdatei „dmlib_filter.conf“

Das DWARF Debugging Information speichert unterschiedslos Informationen über alle Files, aus denen sich das gelinkte Executable zusammensetzt. Insbesondere werden auch Informationen über die Variablen und Unterprogramme sämtlicher eingebundener Libraries angelegt. Bei der Analyse des Cacheverhaltens einer Anwendung sind diese natürlich uninteressant. Deshalb wird die Konfigurationsdatei „dmlib_filter.conf“ bereitgestellt. Sie ermöglicht es, eine Liste von Compile Units anzulegen, deren Debugging Information nicht ausgewertet werden. Bei der Initialisierung der data_mapping_lib werden die zur jeweiligen CU gehörigen DIE-Bäume einfach übersprungen. Um eine Liste aller CUs, die ausgefiltert werden sollen, anzulegen, muss man das instrumentierte Executable mit der Option „–dmlib_print_CUs“ aufrufen. Die „dmlib_init_lib“ gibt dann nur den Namen aller CUs, aus denen sich das Executable zusammemsetzt, auf der Standardausgabe aus und beendet das Programm. Den Namen der CUs, die nicht interessieren, kopiert man dann in eine Datei mit dem Namen „dmlib_filter.conf“, die sich im selben Verzeichnis, wie das instrumentierte Programm befinden muss.

7.4 Debugfunktionen

Die „data_mapping_lib“ stellt einige Debugfunktionen bereit, die interssant sind, wenn ihr Code umgeschrieben werden soll. Zunächst können die globalen Strukturen ausgedruckt werden:

```
int dmlib_print_list_of_vars();
int dmlib_print_little_list_of_vars();
```

Die erste Funktion druckt alle Felder von „var_t“ aus (also den Namen der Variablen, ihre Anfangs- und Endadresse ihr File_ID und Line_Nr, sowie den Zeiger auf das nächste Listenelement). Diese Liste hat häufig auf dem Monitor keinen Platz mehr. Deshalb druckt die zweite Version nur den Namen der Variablen und ihre Anfangsadresse.

```
int dmlib_print_table_of_subprograms();
```

Diese Funktion gibt die „table_of_subprogramms“ auf der Standardausgabe aus.

```
int dmlib_print_tables_of_form_param_and_loc_vars
     (int startindex,int endindex);
```

Jedem Eintrag in die „table_of_subprogramms“ ist eine „table_of_form_param-_and_loc_vars“ zugeordnet. Es kann daher etwas umfangreich werden, diese Tabellen alle auszudrucken. Deshalb gibt es die Möglichkeit, nur die Tabellen auszudrucken, die zu einer „table_of_subprogramms“ mit einem Index zwischen start- und endindex gehören.

Wenn mit dmlib_get_name() bestimmt wurde, welcher Variablenname zu einer Adresse gehört, können mit dmlib_get_range() auch die übrigen Adressen, die zu dieser Variablen gehören, ermittelt werden.

```
int dmlib_print_range(char* name,int file_id,int line_nr);
```

name Der Name der Variablen, deren Adressbereich gesucht wird.

```
int dmlib_print_name(long unsigned address, int file_id,int line_nr);
```

formale Parameter

address Eine beliebige Adresse, die zu der gesuchten Variablen gehrt.

file_id,line_nr Es sind die Werte aus dmlib_entered() des jeweiligen Unterprogramms anzugeben. Wenn beide Parameter den Wert 0 haben, wird nach statischen Variablen gesucht.

Kapitel 8

Zusammenfassung und Ausblick

8.1 Zusammenfassung

Dem zunehmenden Abstand zwischen der stark anwachsenden Prozessorgeschwindigkeit und der schwächer anwachsenden Speicherzugriffsgeschwindigkeit wird schon seit langem mit den Einbau von Caches zwischen CPU und Hauptspeicher begegnet (vgl. Kapitel 2). Da dieser Unterschied immer größer wird, muss die Anzahl der Cachefehlzugriffe immer weiter gesenkt werden bzw. die Lokalität der Datenzugriffe immer weiter verbessert werden. In Kapitel 3 wurden verschiedene Techniken dazu vorgestellt. Im Verhältnis zu den Anforderungen dieser Techniken weisen die gegenwärtigen Performance Measurement Tools allerdings einen großen Mangel auf: sie können nur darstellen, wie oft sich ein (Cache-)Event eines gewissen Typs in einem bestimmten Programmabschnitt ereignet; aber sie können dem Event nicht die Variable, die es ausgelöst hat, zuordnen. Diese Situation ist für den Programmierer sehr unbefriedigend; er braucht den Namen einer Variablen, um die Performance seines Programms verbessern zu können. Die verschiedenen Techniken auf Kapitel 3 zur Optimierung der Datenlokalität beruhen alle entweder auf einer Veränderung des Datenlayouts oder des Datenzugiffs. Die Lösung dieses Problems zerfällt in zwei Teile: Der erste Schritt besteht darin, dass das Performance Measurement Tool die Möglichkeit einer Hardwarekomponente ausnutzt und bei einem Event auch dessen auslösende Adresse beachtet. Diese Feature bieten z. Z. nur der Pentium 4 und der Itanium. Eine Kleinigkeit stört allerdings noch: die Variable liegt in Form einer Adresse und nicht in Form eines Namens vor. Die Aufgabe dieser Diplomarbeit bestand in deren Beseitigung. Es sollte eine Bibliothek erstellt werden, die zu einem bestimmten Zeitpunkt der Programmausführung einer Speicheradresse den Namen der Variablen zuordnet, die sie enthält. Was die statischen bzw. glo-

balen sowie die Stack-Variablen betrifft, ist dies auch gelungen (vgl. jedoch 8.2). Damit kann dann auch der zweite Schritt gemacht werden: Wenn das Performance Measurement Tool diese Bibliothek einbindet, kann das gesamt Problem gelöst werden.
Allerdings ist hierbei gleich wieder eine neue Schwierigkeit entstanden: Wenn die (Cache-)Events nach den Variablen, die sie ausgelöst haben, unterschieden werden, ist die Größe der Daten, die gespeichert werden muss, beträchtlich. Um zu verhindern, dass bei der Analyse des Cacheverhalten eines Programms sehr viele Daten entstehen, ist es also notwendig, sich darauf zu beschränken, nur einige ausgewählte Variablen zu beobachten. Die Lösung dieses Problems setzt die Existenz einer Hardwarekompente voraus, die so konfiguriert werden kann, dass sie nur noch Events wahrnimmt, die von ausgewählten Adressen ausgelöst werden. Außerdem ist eine Kontrollkomponente notwendig, die die Anforderungen des Performance Measurement Tools entgegennimmt. Bevor sie sie an die Hardwarekomponente weiterleiten kann, ist ein kleiner Zwischenschritt notwendig: die Kontrollkomponente erhält Variablen in Form von Namen und muss sie in Form von Adressen weiterreichen. Diese Richtung der Übersetzung ermöglicht die data_mapping_lib ebenfalls. Wenn sie also in die Kontrollkomponente eingebettet werden würde, wäre auch dieses Problem gelöst.

8.2 Zukünftige Entwicklungen

Hardwaremonitore

Sowie ein Hardware-Monitor, wie er vom EP-Cache-Projekt [1] vorgeschlagen wird, exestiert, sollte die Einbettung der data_mapping_lib in dessen Kontrollkomponente getestet werden.

Erweiterung der „list_of_vars“ um die dynamischen Variablen

Man kann drei verschiedene Klassen von Variablen unterscheiden: statische, Stack- und dynamische Variablen. Nur die ersten beiden Klassen sind in der data_mapping_lib implementiert. Falls in Zukunft auch die dynamischen Variablen in die list_of_vars aufgenommen werden, sind die Hinweise auf Seite 30 zu berücksichtigen.

Testen der data_mapping_lib

Die data_mapping_lib ist nie richtig getestet worden. Sie wurde immer nur bei sehr kleinen C-Testprogrammen angewandt. Aber nie bei richtigen App-

lications und nie mit Fortran, Modula oder C++-Programmen (die ja von DWARF2 unterstützt werden).

Unterstützung anderer Debugformate und anderer Plattformen

Neben DWARF2 könnte die data_mapping_lib auch andere andere Debugformate, wie z. B. STUBS unterstützen; und auch andere Plattformen als x86-Rechner, insbesondere 64-Bit Arichitekturen.

8.2.1 Erweiterung der Schnittstelle

Die könnte andere Funktionen bereitstellen, z.B. z. B. eine Abbildung von der Speicheradresse auf den Datentyp(was zur Datenreduktion nützlich sein kann) oder das Erfragen des Indices einer (evtl. mehrdimensionalen) Array-Variablen.

Anhang A

Debugging und das DWARF-Format

A.1 Debugging Information

Um die Ausführung eines Programms in einer für den Menschen verständlchen Weise sichtbar machen zu können, sind gewiße Informationen aus den Quellcode des Programms notwendig. Diese müßen beim Compiliervorgang angelegt werden und in einer Form gespeichert werden, daß später auf diese Informationen zugegriffen werden kann. Dazu müssen die Hersteller von Compilern und Debuggern sich auf ein einheitliches Format von Codierung und Darstellung dieser Daten geeinigt haben. Das bekannteste ist das *DWARF debugging information format*. Es soll im Folgendem vorgestellt werden.

A.2 DWARF Version 1, 2 und 3

Die Version 1 von DWARF wurde am 20.1.92 verffentlicht. Schon kurze Zeit später – am 27.7.1993 – folgte die Version 2 von DWARF (Revision: 2.0.0). Der Inhalt der Debug-Information ist zwar –von einigen Erweiterungen abgesehen– gleich geblieben; Version 2 ermöglicht aber eine wesentlich kompakter Speicherung. Ein wesentliches Ziel von DWARF ist es, Debug-Informationen bereitzustellen, die jede beliebige Programmiersprache beschreiben können. In der Spezifikation von DWARF2 wird trotzdem ausdrücklich darauf hingewiesen, daß die Bedürfnisse von C, C++, Fortran77, Fortran90, Modula2 und Pascal abgedeckt werden. Die Version 3 von DWARF wurde vom 'Dwarf Standards Committee' zwischen Nov 1999 und Juli 2001 entwickelt. Die Änderungen von DWARF2 zu DWARF3 betreffen diesmal aber nicht die Art der Speicherung des Formats; es wird deshalb an-

genommen, daß der Versionswechsel nicht schwer fällt. Zu den Neuerungen gehören, eine Unterstütztung der Programmiersprache Java.

A.3 Allgemeine Ziele von DWARF

Ziel war eine symbolische, quellcodeorientierte, aber sprachunabhängige Darstellung von Aufbau und Daten eines Programms. Mit Hilfe von Objekten, die für alle Programmiersprachen gültig sind, soll es Debuggern ermöglicht werden, sich ein genaues Bild des Quellprogramms zu machen und dadurch das Executable zur Laufzeit in einer menschlich lesbaren Form darstellen zu können. Des Format erfüllt folgende Bedingungen:

1. *Freiheit des Design eines Compilers oder Debuggers*
   Die Debugging-Informationen werden in einer Form zugänglich gemacht, die keine bestimmte Implementierung von Compilern oder Debuggern notwendig macht.

2. *Unabhänigkeit von einer bestimmten Sprache*
   Die Debug-Informationseinheiten werden möglichst allgemein definiert. Ähnliche Bestandteile verschiedener Sprachen werden von derselben Informationseinheit beschrieben. Nur bei einzigartigen Konzepten (z.B. Fortans 'common blocks') werden Attribute definiert, die speziell auf eine Sprache zugeschnitten sind.

3. *Erweiterbarkeit*
   Herstellern von Compilern und Debuggern wird die Möglichkeit gegeben, die DWARF -Spezifikation um eigene Bestandteile zu erweitern. Dies ist z.B. nötig, wenn neue Programmiersprachen wichtig werden, deren Konzepte mit den bisherigen Mitteln nicht adäquat beschrieben werden können. Von dieser Möglichkeit wird intensiv Gebrauch gemacht.

4. *Abwärtskompatibilität*
   Sogar die Version 3 von DWARF ist im wesentlichen abwärtskompatibel zur Version 2.

A.4 Die verschiedenen Sektion von DWARF2

DWARF2 speichert seine Debugging Information in neun verschiedenen Abschnitten des Objektfiles unter den Namen: der *„.debug_abbrev“*, der *„.debug_aranges“*, der *„.debug_frame“*, der *„.debug_info“*, der *„.debug_line“*, der

„.debug_loc“, der *„.debug_macinfo“*, der *„.debug_pubnames“* und der *„.debug_str“*. Einige werden im Folgendem vorgestellt, vor allem die *„.debug_info“*-Sektion, die die DIEs(„Debugging Information Entries“) enthält. Hersteller von Compilern und Debuggern haben weiter Sektionen hinzugefügt, z.B. hat SGI die *„.debug_weaknames“*, die *„.debug_funcnames“*, die *„.debug_typenames“* und die *„.debug_varnames“* eingeführt. Die '.debug_typenames' wurde in DWARF3 sinngemäß übernommen und in '.debug_pubtypes' umbenannt.

A.5 Die '.debug_info'-Sektion

A.5.1 DIEs,Tags und Attribute

Die elementare Einheit, durch die das Quellprogramm beschreiben wird, wird in DWARF *Debugging Information Entry (DIE)* genannt. Jeder DIE ist mit einem **tag** versehen, das angibt, zu welcher Klasse der DIE gehört. Mögliche Klassen sind in aufgeführt. Die Klassen der DIEs können in drei grundsätzliche Kathergorien unterteilt werden: Solche, die den Progeammscope, solche, die Datenobjekte und solche, die Typdefinitionen beschreiben. Außerdem gehören zu einer DIE ein oder mehrere Attribute. Attribute sind Paare von Name und Wert. Die Attribute geben die speziellen Eigenschaften einer DIE einer bestimmten Klasse an. Z.B. gehören zu einer DIE mit dem Tag „DW_TAG_variable“ u.a. die Attribute mit dem Namen „DW_AT_name“, „DW_AT_location“ und „DW_AT_byte_size“.

A.5.2 DIE-Bäume

Um hierarchische Abhänigkeiten ausdrücken zu können, darf jeder DIE ein oder mehrere Kinder haben. Umgekehrt muß jeder DIE (außer einer) Kind eines anderen DIEs sein. Auf diese Weise ensteht eine Baumstruktur mit den DIEs als Knoten. Zur Speicherung wird der DIE-Baum in Präfixordnung flachgemacht. Damit die Baumstruktur rekonstruiert werden kann, muß in jedem DIE gespeichert sein, ob er (mindestens) ein Kind hat oder nicht. Wenn er kein Kind hat und auf ihn kein NULL-Eintrag folgt, ist der nächste DIE ein Geschwister. Wenn er ein Kind hat, ist der nächste DIE das erste Kind. Ob er noch weitere Kinder hat, hängt davon ab, ob das erste Kind Geschwister hat. Eine Folge von Geschwister-DIEs wird durch einen NULL-Eintrag beendet.

A.5.3 Referenzen

Für den Anwender der Debug-Informationen ist diese Praefix-Ordnung häufig nicht sehr brauchbar; er ist u.U. daran interessiert von einen DIE-Knoten direkt (also unter Umgehung der Baumstruktur) zu einem andern zu springen. Deshalb können die DIEs Attribute haben, deren Werte Referenzen aus andere DIEs sind. Z.B. kann einem das Attribut „DW_AT_sibling“ zugewiesen werden. Dadurch können in einer for-Schleife alle Geschwisterknoten einer DIE durchlaufen werden. Wegen der Referenzen kann die Zusammenhang der DIE-Knoten genaugenommen nicht mehr durch einen Baum beschrieben werden.

A.5.4 „Compile Units“

Eine „Compile Unit“ (CU) ist jede relokierbare Objektdatei, aus dem das Executable beim Linken zusammengesetzt wird. Also z.B. ist jede Datei mit dem Hauptprogramm oder einem Unterprogramm sowie jede Library eine CU. Jeder CU ist ein DIE-Baum zugeordnet. An der Wurzel dieses Baumes ist befindet sich ein DIE mit dem Tag „DW_TAG_compile_unit“.
Wichtige Attribute dieses TAGs sind:

DW_AT_low_pc Der Wert dieses Attributs ist die erste relokierte Addresse des Mashinencodes, der zu der CU gehört.

DW_AT_high_pc Der Wert dieses Attributs ist die erste relokierte Addresse nach dem Mashinencode, der zu der CU gehört.

DW_AT_name Der Wert dieses Attributs ist der relative oder absolute Pfad des Sourcefiles, von der die CU abgeleitet wurde.

DW_AT_language Der Wert dieses Attributs ist ein Code für die Programmiersprache in der der Quellcode der CU gechrieben wurde. In DWARF2 sind vorgesehen:
- DW_LANG_C (Non-ANSI C)
- DW_LANG_C89 (ISO/ANSI C)
- DW_LANG_C_plus_plus (C++)
- DW_LANG_Fortran77 (Fortran77)
- DW_LANG_Fortran90 (Fortran90)
- DW_LANG_Modula2 (Modula2)
- DW_LANG_Pascal83 (ISO/ANSI Pascal)

A.6 Die '.debug_pubnames', '.debug_pubtypes' und '.debug_aranges'-Sektionen

Ein Debugger steht häufig vor der Aufgabe, Informationen über globale Objekte zu finden, die nicht in der Compile Unit definiert sind, in der der Debugger angehalten wurde. Globale Objekte sind Datenobjekte oder Unterprogramme oder Typdefinitionen, die sich innerhalb des äußersten Sichtbarkeitsbereich einer 'Compile Unit' befinden. Der Debugger weiß entweder ihre Adresse oder ihren Namen. In beiden Fällen müssten i.a. alle 'DIE'-Bäume durchsucht werden. Falls die Adresse bekannt ist, können zumindest die Attribut 'DW_AT_low_pc' und 'DW_AT_high_pc' benutzt werden, um die Suche abzukürzen und den richtigen 'DIE'-Baum zu finden. Dann müsen aber alle in Frage kommenden DIEs untersucht werden. Falls nur der Name bekannt ist, muß jeder 'DIE'-Baum durchsucht werden, solange bis das Objekt gefunden ist. Außerdem müßte der Debugger bei seiner Suche durch die DIE-Bäume verschiedener CUs i.a. auf mehrere Speicherseiten zugreifen, was -insbesondere bei großen Programmen- seine Performance deutlich senken würde. Es wäre also nicht sehr effizient, die DIE-Bäume der verschiedenen CUs zu durchsuchen, um Informationen über ein globales Objekt zu finden. Deshalb ermöglicht es DWARFT zwei oder drei Tabellen zu definieren, die den Zugriff auf globale Objekte beschleunigen. Für den Zugriff mit Hilfe des Namens ist die '.debug_pubnames' vorgesehen; in der Version 3 kommt noch die '.debug_pubtypes' dazu. Für den Zugriff mit Hilfe der Addresse ist die '.debug_aranges' zuständig. Diese Tabellen enthalten den Offset der 'Compile Unit' des globalen Objekts innerhalb der '.debug_info'-Sektion und den Offset seiner DIE innerhalb seiner CU.

A.7 Die '.debug_line'-Sektion

Debugger arbeiten häufig mit den Begriffen des Quellprogramms, als wäre es ein Binäry. Insbesondere können sie das Quellprogramm Zeile für Zeile durchlaufen. Dazu ist es nötig zu wissen, welche Maschinenbefehle einer Zeile des Quellprogramms entsprechen. Die '.debug_info'-Sektion enthält nur die Struktur des Executable. Deshalb werden diese Informationen in einer eigenen Sektion des Objektfiles, nämlich der '.debug_line'-Sektion gespeichert.

A.8 Die '.debug_macinfo'-Sektion

Einige Programmiersprachen, z.B. C, ermöglichen es, Text aus dem Quellcode durch Makros zu ersetzten. Diese Ersetztungen werden schon vom Präprozeßor vollzogen. Zum Zeitpunkt der Compilation existieren keine Makros mehr; sie sind kein Bestandteil der eigentlichen Programmiersprache. Deshalb beziehen sich die DWARF Debugging Information auf den expandierten Quellcode. Um troztdem Information über Makros bereitstellen zu können, gibt es die „.debug_macinfo"-Sektion.

A.9 Die Libdwarf

Ein sehr nützliches Werkzeug, um auf die Debugging Information des DWARF-Formats zuzugreifen, ist die *„libdwarf"*. Sie wird von *silicon grafics* zum Programmieren ihrer Debbugger entwickelt und ist seit etwas über einem Jahr als Open-source im Internet erhältlich. Sie macht die verschiedenen DWARF-Sektionen zugänglich. Z.b. kann innerhalb der *„.debug_info"*-Sektion von DIE-Baum einer CU zum DIE-Baum der nächsten CU gesprungen werden. Innerhalb eines DIE-Baums kann man navigieren; sich die *tags* der Knoten, und die Namen und Werte ihrer Attribute ansehen.

Literaturverzeichnis

[1] http://www.scai.fhg.de/EP-Cache

[2] http://h30097.www3.hp.com/dcpi/

[3] Homepage: http://icl.cs.utk.edu/papi/
Download: http://icl.cs.utk.edu/[..]jects/papi/downloads/

[4] Homepage: http://www.csd.uu.se/ mikpe/linux/perfctr/

[5] manpage:http://techpubs.sgi.com/

[6] Download: http://reality.sgi.com/davea/#dwarf

[7] Homepage: http://coldstore.sourceforge.net/
Download: http://sourceforge.net/projects/coldstore/