- Was können optimierende Übersetzer erreichen?
- Wie lassen sich optimierende Übersetzer unterstützen?
- Welche Fallen können sich durch den Einsatz von optimierenden Übersetzer eröffnen?

Was wird optimiert?

Es gibt zwei teilweise gegensätzliche Ziele der Optimierung:

- Minimierung der Länge des erzeugten Maschinencodes.
- Minimierung der Ausführungszeit.

Es ist relativ leicht, sich dem ersten Ziel zu nähern. Die zweite Problemstellung ist in ihrer allgemeinen Form nicht vorbestimmbar (wegen potentiell unterschiedlicher Eingaben) bzw. in seiner allgemeinen Form nicht berechenbar.

- Die Problemstellung ist grundsätzlich für sehr kleine Sequenzen lösbar.
- Bei größeren Sequenzen wird zwar nicht das Minimum erreicht, dennoch sind die Ergebnisse beachtlich, wenn alle bekannten Techniken konsequent eingesetzt werden.

- Der GNU-Superoptimizer generiert sukzessive alle möglichen Instruktionssequenzen, bis eine gefunden wird, die die gewünschte Funktionalität umsetzt.
- Die Überprüfung erfolgt durch umfangreiche Tests, ist aber kein Beweis, dass die gefundene Sequenz äquivalent zur gewünschten Funktion ist. In der Praxis sind jedoch noch keine falschen Lösungen geliefert worden.
- Der Aufwand des GNU-Superoptimizers liegt bei $O((mn)^{2n})$, wobei m die Zahl der zur Verfügung stehenden Instruktionen ist und n die Länge der kürzesten Sequenz.
- Siehe `https://ftp.gnu.org/gnu/superopt/`
  (ist von 1995 und lässt sich leider mit modernen C-Übersetzern nicht mehr übersetzen).

- Problemstellung: Gegeben seien zwei nicht-negative ganze Zahlen in den Registern r_1 und r_2. Gewünscht ist das Minimum der beiden Zahlen in r_1.
- Eine naive Umsetzung erledigt dies analog zu einer **if**-Anweisung mit einem Vergleichstest und einem Sprung.
- Folgendes Beispiel zeigt dies für die SPARC-Architektur und den Registern %l0 und %l1:

```
        subcc   %l0,%l1,%g0
        bleu    endif
        nop
        or      %l1,%g0,%l0
endif:
```

```
        subcc   %l1,%l0,%g1
        subx    %g0,%g0,%g2
        and     %g2,%g1,%l1
        addcc   %l1,%l0,%l0
```

- Der Superoptimizer benötigt (auf Theon) nur eine Sekunde, um 28 Sequenzen mit jeweils 4 Instruktionen vorzuschlagen, die allesamt das Minimum bestimmen, ohne einen Sprung zu benötigen. Dies ist eine der gefundenen Varianten.
- Die Instruktionen entsprechen folgendem Pseudo-Code:

```
%g1 = %l1 - %l0
carry = %l0 > %l1? 1: 0
%g2 = -carry
%l1 = %g2 & %g1
%l0 = %l1 + %l0
```

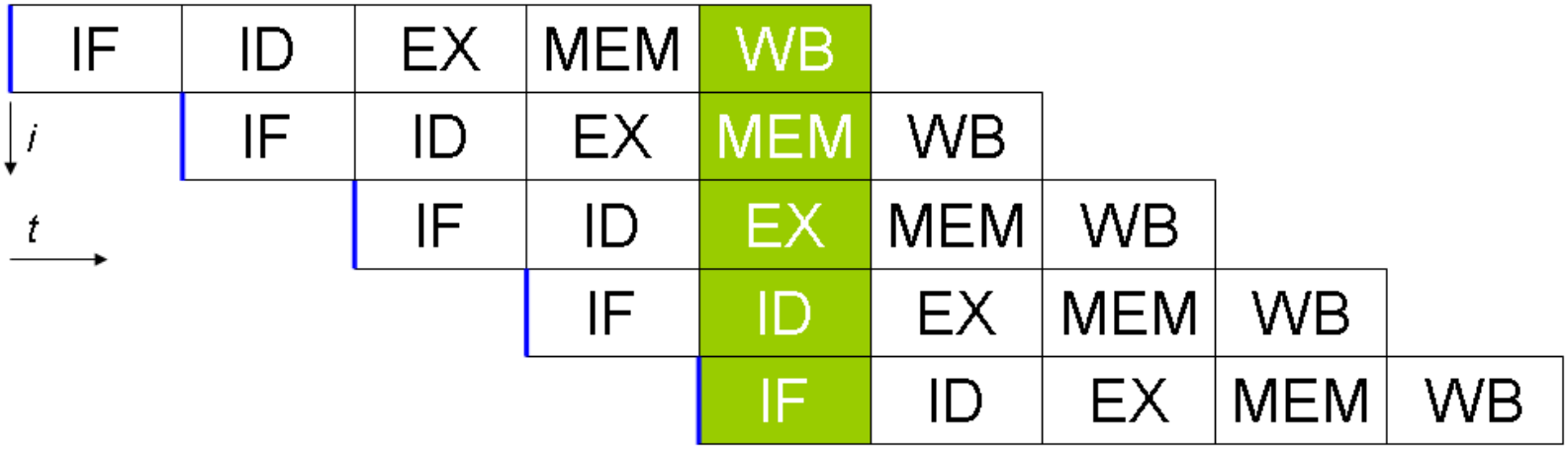
- Generell ist die Vermeidung bedingter Sprünge ein Gewinn, da diese das Pipelining erschweren.

- Moderne Prozessoren arbeiten nach dem Fließbandprinzip: Über das Fließband kommen laufend neue Instruktionen hinzu und jede Instruktion wird nacheinander von verschiedenen Fließbandarbeitern bearbeitet.
- Dies parallelisiert die Ausführung, da unter günstigen Umständen alle Fließbandarbeiter gleichzeitig etwas tun können.
- Eine der ersten Pipelining-Architekturen war die IBM 7094 aus der Mitte der 60er-Jahre mit zwei Stationen am Fließband. Die UltraSPARC-IV-Architektur hat 14 Stationen, die Broadwell-Mikroarchitektur hat 16.
- Die RISC-Architekturen (RISC = *reduced instruction set computer*) wurden speziell entwickelt, um das Potential für Pipelining zu vergrößern.
- Bei der Pentium-Architektur werden im Rahmen des Pipelinings die Instruktionen zuerst intern in RISC-Instruktionen konvertiert, so dass sie ebenfalls von diesem Potential profitieren kann.

Um zu verstehen, was alles innerhalb einer Pipeline zu erledigen ist, hilft ein Blick auf die möglichen Typen von Instruktionen:

- Operationen, die nur auf Registern angewendet werden und die das Ergebnis in einem Register ablegen (wie etwa subcc in den Beispielen).
- Instruktionen mit Speicherzugriff. Hier wird eine Speicheradresse berechnet und dann erfolgt entweder eine Lese- oder eine Schreiboperation.
- Sprünge.

Zerlegung in Arbeitsschritte



Eine einfache Aufteilung sieht folgende einzelne Schritte vor:

- Instruktion vom Speicher laden (IF)
- Instruktion dekodieren (ID)
- Instruktion ausführen, beispielsweise eine arithmetische Operation oder die Berechnung einer Speicheradresse (EX)
- Lese- oder Schreibzugriff auf den Speicher (MEM)
- Abspeichern des Ergebnisses in Registern (WB)

Das Diagramm wurde von Stephane Tsacas geklaut.

- Bedingte Sprünge sind ein Problem für das Pipelining, da unklar ist, wie gesprungen wird, bevor es zur Ausführungsphase kommt.
- RISC-Maschinen führen typischerweise die Instruktion unmittelbar nach einem bedingten Sprung immer mit aus, selbst wenn der Sprung genommen wird. Dies mildert etwas den negativen Effekt für die Pipeline.
- Im übrigen gibt es die Technik der *branch prediction*, bei der ein Ergebnis angenommen wird und dann das Fließband auf den Verdacht hin weiterarbeitet, dass die Vorhersage zutrifft. Im Falle eines Misserfolgs muss dann u.U. recht viel rückgängig gemacht werden.
- Das ist machbar, solange nur Register verändert werden. Manche Architekturen verfolgen die Alternativen sogar parallel und haben für jedes abstrakte Register mehrere implementierte Register, die die Werte für die einzelnen Fälle enthalten.
- Die Vorhersage wird vom Übersetzer generiert. Typisch ist beispielsweise, dass bei Schleifen eine Fortsetzung der Schleife vorhergesagt wird.

Es gibt zwei Möglichkeiten, um die Vorhersage bei Branch Prediction zu verbessern:

- Durch Profiling: Die Option „-fprofile-arcs“ beim *gcc* instrumentiert den Code, so dass Statistiken über den Verlauf zur Laufzeit generiert werden in Dateien, die in „.gcda“ enden. Diese Daten können mit der Option „-fbranch-probabilities“ bei einer späteren Neu-Übersetzung genutzt werden, um die Vorhersagen des Übersetzers zu verbessern.
- Durch explizite Vorgaben: Der *gcc* unterstützt die Builtin-Funktion *__builtin_expect*, die vom Übersetzer als Vorhersage übernommen wird. Beispiel:

```
ptr = head;
while (__builtin_expect(ptr != 0, 1)) {
    /* ... */
    ptr = ptr->next;
}
```

Normalerweise empfiehlt sich der erstere Ansatz.

- Bei moderneren Architekturen mit sehr langen Pipelines kommt ein statischer Hinweis viel zu spät, da hierzu die Instruktion bereits dekodiert sein muss.
- Neuere Architekturen verlassen sich daher ausschließlich auf ihre eigenen Vorhersagen auf der Basis früheren Verhaltens.
- Offenbar kommt beginnend mit der 2013 erschienenen Haswell-Architektur von Intel kommt das TAGE-Verfahren zum Einsatz, das 2006 von André Seznec und Pierre Michaud veröffentlicht worden ist: *A case for (partially) TAgged GEometric history length branch prediction*
- Dieses Verfahren zieht die vorangegangene Sprunghistorie und die Adresse der Sprung-Instruktion in Betracht und verwendet in der zugehörigen Hash-Tabelle partielle Tags, um im Falle von Kollisionen falsche Entscheidungen zu vermeiden.
- Statische Hinweise wie der entsprechende x86-Instruktionspräfix werden seit dem Aufkommen dynamischer Verfahren ignoriert.

Typische Techniken zur Minimierung des Maschinencodes

- Lokale Variablen und Parameter soweit wie möglich in Registern halten. Dies spart Lade- und Speicherinstruktionen.
- Vereinfachung von Blattfunktionen. Das sind Funktionen, die keine weitere Funktionen aufrufen.
- Auswerten von konstanten Ausdrücken während der Übersetzzeit (*constant folding*).
- Vermeidung von Sprüngen.
- Vermeidung von Multiplikationen, wenn einer der Operanden konstant ist.
- Elimination mehrfach vorkommender Teilausdrücke.
  Beispiel: `a[i + j] = 3 * a[i + j] + 1;`
- Konvertierung absoluter Adressberechnungen in relative.
  Beispiel: `for (int i = 0; i < 10; ++i) a[i] = 0;`
- Datenflussanalyse und Eliminierung unbenötigten Programmtexts

- Ein Übersetzer kann lokale Variablen nur dann permanent in einem Register unterbringen, wenn zu keinem Zeitpunkt eine Speicheradresse benötigt wird.
- Sobald der Adress-Operator & zum Einsatz kommt, muss diese Variable zwingend im Speicher gehalten werden.
- Das gleiche gilt, wenn Referenzen auf die Variable existieren.
- Zwar kann der Übersetzer den Wert dieser Variablen ggf. in einem Register vorhalten. Jedoch muss in verschiedenen Situationen der Wert neu geladen werden, z.B. wenn ein weiterer Funktionsaufruf erfolgt, bei dem ein Zugriff über den Zeiger bzw. die Referenz erfolgen könnte.

```
int index;
while (scanf("%d", &index) == 1) {
   a[index] = f(a[index]);
   a[index] += g(a[index]);
}
```

- In diesem Beispiel wird ein Zeiger auf *index* an die *scanf*-Funktion übergeben. Der Übersetzer weiß nicht, ob diese Adresse über irgendwelche Datenstrukturen so abgelegt wird, dass die Funktionen *f* und *g* darauf zugreifen. Entsprechend wird der Übersetzer genötigt, immer wieder den Wert von *index* aus dem Speicher zu laden.
- Deswegen ist es ggf. hilfreich, explizit eine weitere lokale Variablen zu verwenden, die eine Kopie des Werts erhält und von der keine Adresse genommen wird:

```
int index;
while (scanf("%d", &index) == 1) {
   int i = index;
   a[i] = f(a[i]);
   a[i] += g(a[i]);
}
```

```
bool find(int* a, int len, int* index) {
   while (*index < len) {
      if (a[*index] == 0) return true;
      ++*index;
   }
   return false;
}
```

- Parameter, die über einen Zeiger übergeben werden, sollten bei häufiger Nutzung in ausschließlich lokal genutzte Variablen kopiert werden, um einen externen Einfluss auszuschließen.

```
bool find(int* a, int len, int* index) {
   for (int i = *index; i < len; ++i) {
      if (a[i] == 0) {
         index = i; return true;
      }
   }
   *index = len;
   return false;
}
```

```
void f(int* i, int* j) {
   *i = 1;
   *j = 2;
   // value of *i?
}
```

- Wenn mehrere Zeiger oder Referenzen gleichzeitig verwendet werden, unterbleiben Optimierungen, wenn nicht ausgeschlossen werden kann, dass mehrere davon auf das gleiche Objekt zeigen.
- Wenn der Wert von `*i` verändert wird, dann ist unklar, ob sich auch `*j` verändert. Sollte anschließend auf `*j` zugegriffen werden, muss der Wert erneut geladen werden.
- In C gibt es die Möglichkeit, mit Hilfe des Schlüsselworts **restrict** Aliasse auszuschließen:

```
void f(int* restrict i, int* restrict j) {
   *i = 1;
   *j = 2;
   // *i == 1 still assumed
}
```

- Mit der Datenflussanalyse werden Abhängigkeitsgraphen erstellt, die feststellen, welche Variablen unter Umständen in Abhängigkeit welcher anderer Variablen verändert werden können.
- Im einfachsten Falle kann dies auch zur Propagation von Konstanten genutzt werden.
  Beispiel: Nach `int a = 7; int b = a;` ist bekannt, dass b den Wert 7 hat.
- Die Datenflussanalyse kann für eine Variable recht umfassend durchgeführt werden, wenn sie lokal ist und ihre Adresse nie weitergegeben wurde.

```
void loopingsleep(int count) {
   for (int i = 0; i < count; ++i)
      ;
}
```

- Mit Hilfe der Datenflussanalyse lässt sich untersuchen, welche Teile einer Funktion Einfluss haben auf den **return**-Wert oder die außerhalb der Funktion sichtbaren Datenstrukturen.
- Anweisungen, die nichts von außen sichtbares verändern, können eliminiert werden.
- Auf diese Weise verschwindet die **for**-Schleife im obigen Beispiel.
- Der erwünschte Verzögerungseffekt lässt sich retten, indem in der Schleife unter Verwendung der Schleifenvariablen eine externe Funktion aufgerufen wird. (Das funktioniert, weil normalerweise keine globale Datenflussanalyse stattfindet.)

- Globale Variablen können ebenso in die Datenflussanalyse einbezogen werden.
- Bei C wird davon ausgegangen, dass globale Variablen sich nicht überraschend ändern, solange keine Alias-Problematik vorliegt und keine unbekannten Funktionen aufgerufen werden.
- Das ist problematisch, wenn Threads oder Signalbehandler unsynchronisiert auf globale Variablen zugreifen.
- In diesem Fällen muss **volatile** verwendet werden.

Optimierende Übersetzer bieten typischerweise Stufen an. Recht typisch ist dabei folgende Aufteilung des gcc:

Stufe	Option	Vorteile
0		schnelle Übersetzung, mehr Transparenz beim Debugging
1	-O1	lokale Peephole-Optimierungen
2	-O2	Minimierung des Umfangs des generierten Codes
3	-O3	Minimierung der Laufzeit mit ggf. umfangreicheren Code
4	-Ofast	Abweichungen von Standard sind möglich – dies betrifft insbesondere mathematische Anwendungen, da u.a. die *order of evaluation* nicht eingehalten wird

Zusätzlich bietet sich noch das Profiling an und ggf. Optionen wie „-funroll-loops“.

Bei der (oder den) höchsten Optimierungsstufe(n) wird teilweise eine erhebliche Expansion des generierten Codes in Kauf genommen, um Laufzeitvorteile zu erreichen. Die wichtigsten Techniken:

- Loop unrolling
- Instruction scheduling
- Function inlining
- Vektorisierungen

```
for (int i = 0; i < 100; ++i) {
   a[i] = i;
}
```

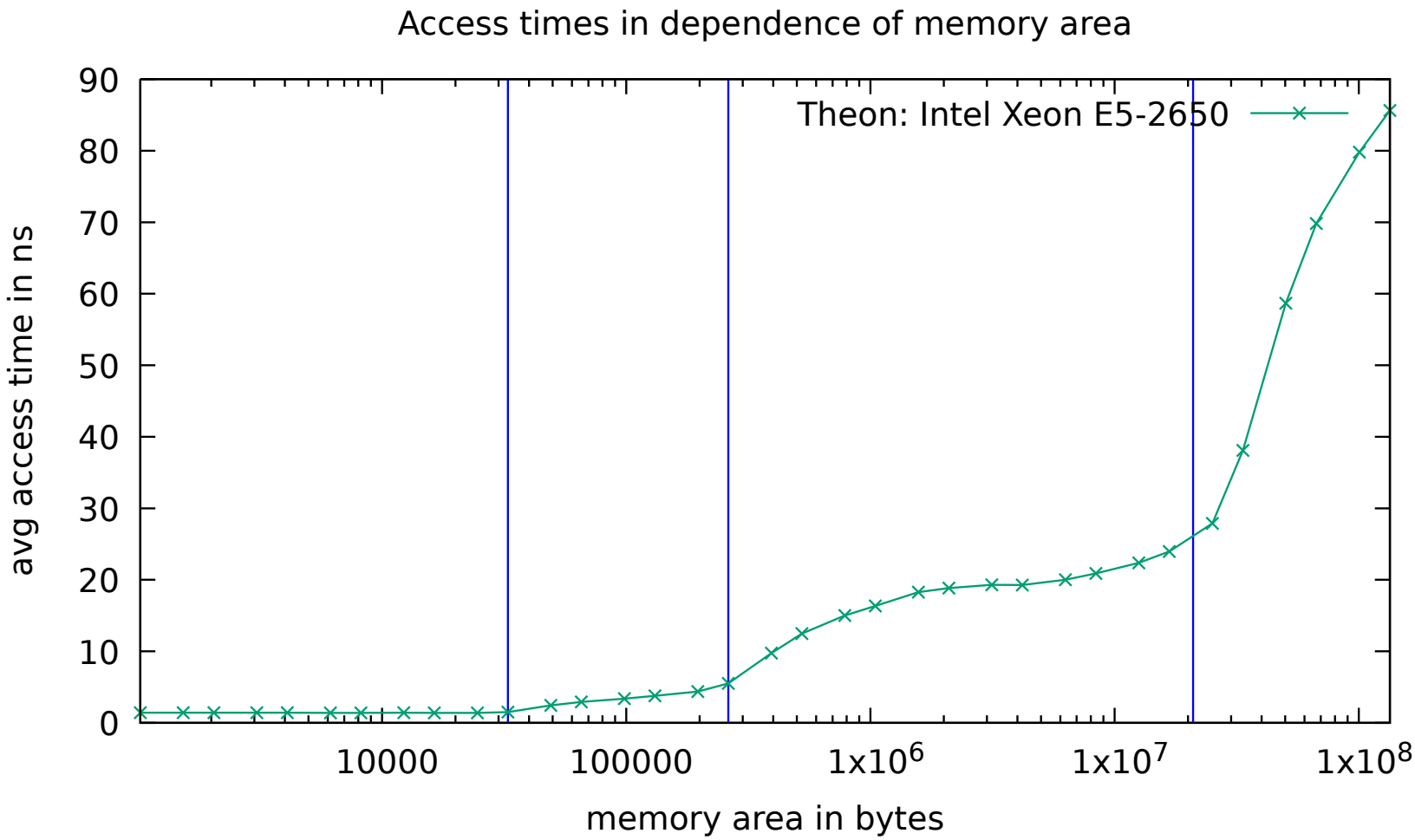
- Diese Technik reduziert deutlich die Zahl der bedingten Sprünge, indem mehrere Schleifendurchläufe in einem Zug erledigt werden.
- Das geht nur, wenn die einzelnen Schleifendurchläufe unabhängig voneinander erfolgen können, d.h. kein Schleifendurchlauf von den Ergebnissen der früheren Durchgänge abhängt.
- Dies wird mit Hilfe der Datenflussanalyse überprüft, wobei sich der Übersetzer auf die Fälle beschränkt, bei denen er sich sicher sein kann. D.h. nicht jede für diese Technik geeignete Schleife wird auch tatsächlich entsprechend optimiert.

```
for (int i = 0; i < 100; i += 4) {
   a[i] = i;
   a[i+1] = i + 1;
   a[i+2] = i + 2;
   a[i+3] = i + 3;
}
```

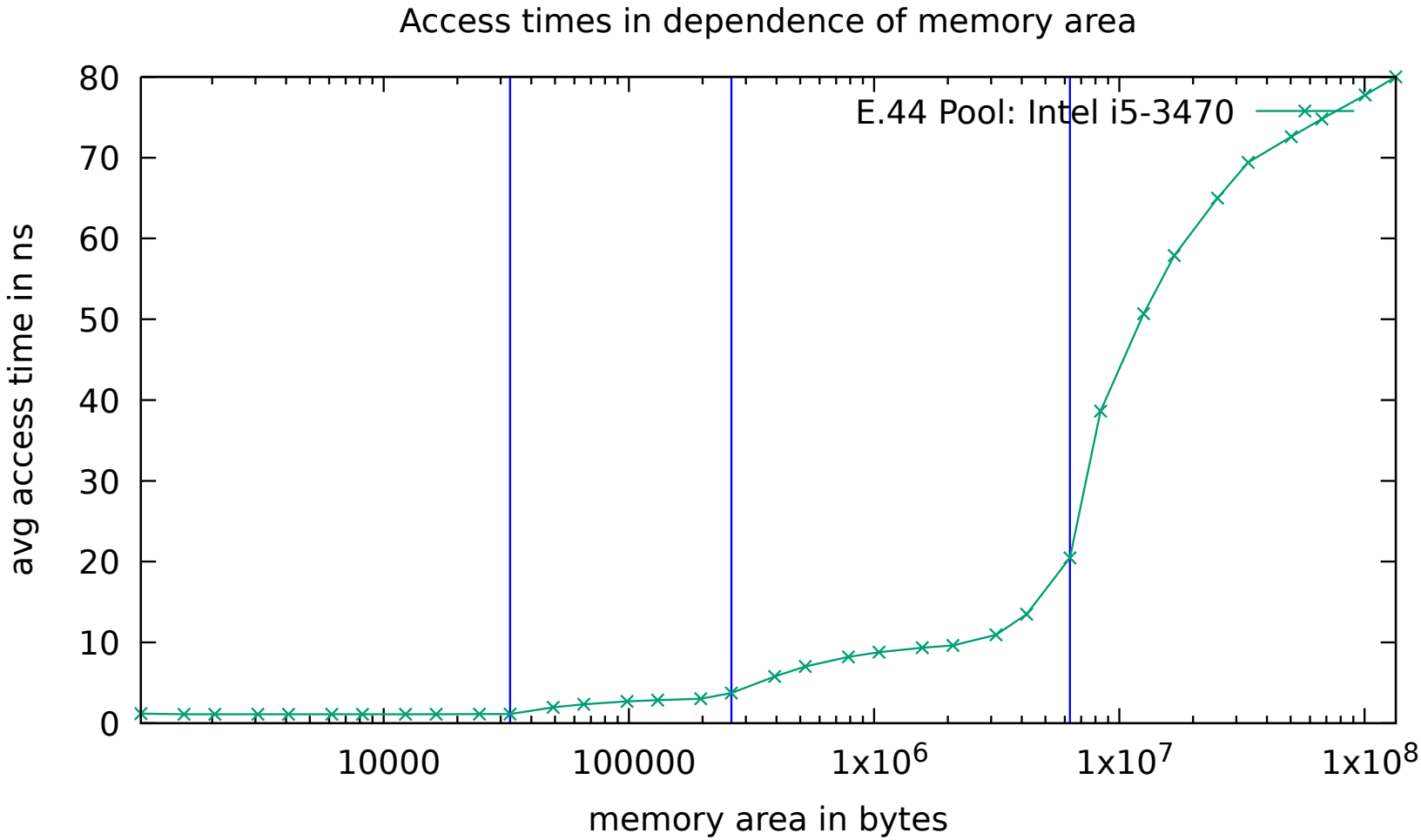
- Zugriffe einer CPU auf den primären Hauptspeicher sind vergleichsweise langsam. Obwohl Hauptspeicher generell schneller wurde, behielten die CPUs ihren Geschwindigkeitsvorsprung.
- Grundsätzlich ist Speicher direkt auf einer CPU deutlich schneller. Jedoch lässt sich Speicher auf einem CPU-Chip aus Komplexitäts-, Produktions- und Kostengründen nicht beliebig ausbauen.
- Deswegen arbeiten moderne Architekturen mit einer Kette hintereinander geschalteter Speicher. Zur Einschätzung der Größenordnung sind hier die Angaben für die Theon, die mit einem E5-2650-v4-Prozessor der Broadwell Mikroarchitektur augestattet ist:

Cache	Kapazität	Taktzyklen
Register		1
L1-Cache	32 KiB	4-5
L2-Cache	256 KiB	12
L3-Cache	20 MiB	38
Hauptspeicher	96 GiB	38 + 58ns

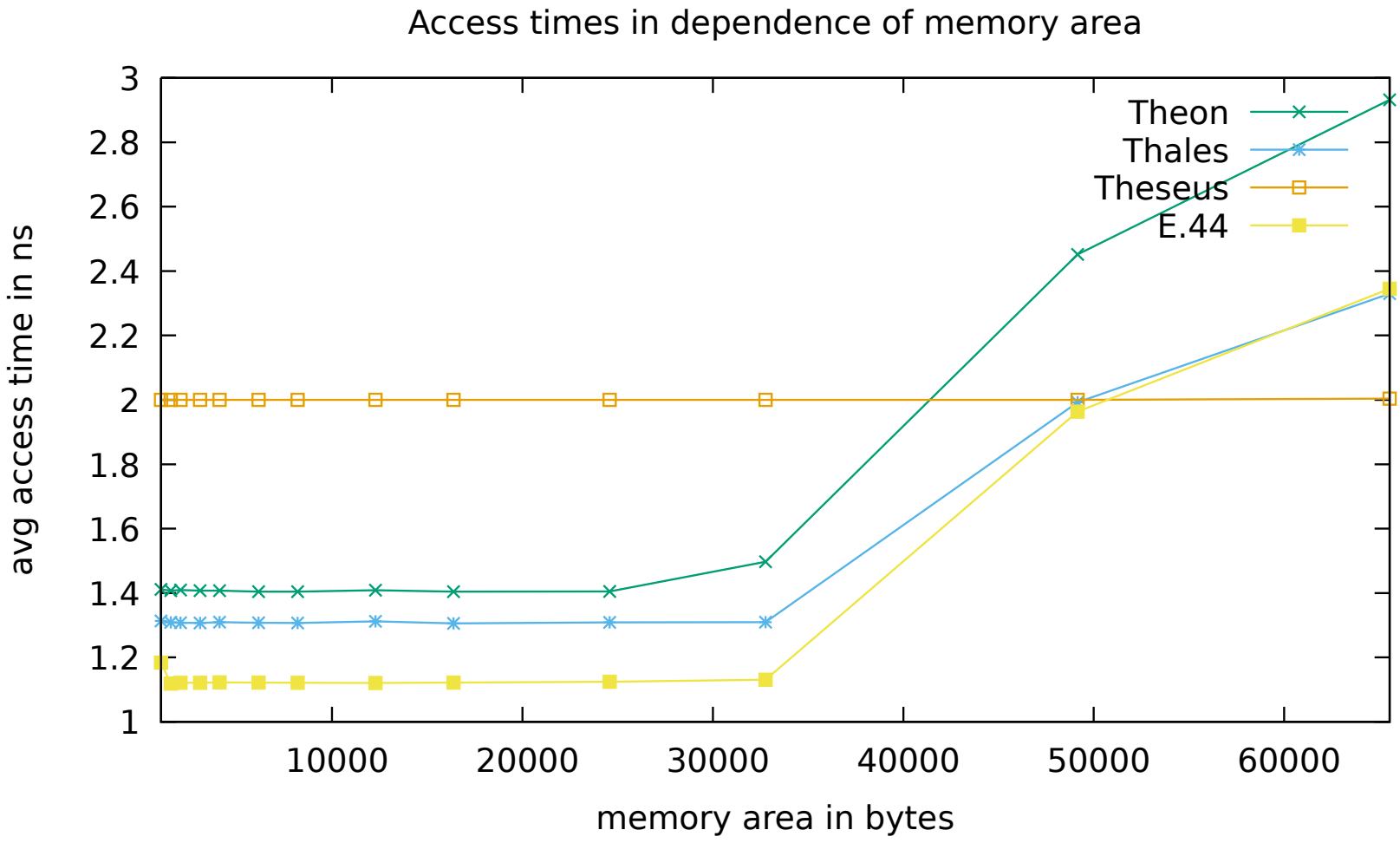
Gemessene Speicherzugriffszeiten: Theon



Theon: 1 Intel E5-2650-Prozessor mit 12 Kernen mit je 2 Threads
Caches: L1 (32 KiB), L2 (256 KiB), L3 (20 MiB)



E.44: 1 Intel i5-3470-Prozessor mit 4 Kernen
Caches: L1 (32 KiB), L2 (256 KiB), L3 (6 MiB)


Access times in dependence of memory area
avg access time in ns
memory area in bytes
Theon
Thales
Theseus
E.44
3
2.8
2.6
2.4
2.2
2
1.8
1.6
1.4
1.2
1
10000
20000
30000
40000
50000
60000

- Ein Cache ist in sogenannten *cache lines* organisiert, d.h. eine *cache line* ist die Einheit, die vom Hauptspeicher geladen oder zurückgeschrieben wird.
- Jede der *cache lines* umfasst – je nach Architektur – 32 - 128 Bytes. Auf der Theseus sind es beispielsweise 64 Bytes.
- Jede der *cache lines* kann unabhängig voneinander gefüllt werden und einem Abschnitt im Hauptspeicher entsprechen.
- Das bedeutet, dass bei einem Zugriff auf `a[i]` mit recht hoher Wahrscheinlichkeit auch `a[i+1]` zur Verfügung steht.
- Entweder sind Caches vollassoziativ (d.h. jede *cache line* kann einen beliebigen Hauptspeicherabschnitt aufnehmen) oder für jeden Hauptspeicherabschnitt gibt es nur eine *cache line*, die in Frage kommt (*fully mapped*), oder jeder Hauptspeicherabschnitt kann in einen von *n* *cache lines* untergebracht werden (*n-way set associative*).

- Es gibt keine portable Möglichkeit, die Cache-Konfiguration zu ermitteln.
- Unter Linux gibt es in der Hierarchie unter */sys/devices/system/cpu/cpu0/cache* für jeden der Caches der *cpu0* ein Verzeichnis *index0*, *index1* usw.
- In jedem Cache-Verzeichnis finden sich folgende Dateien:

level	Level des Cache, typischerweise 1, 2 oder 3
type	„Data“, „Instruction“ oder „Unified“
coherency_line_size	Größe einer *cache line*
ways_of_associativity	wieviele *cache lines* kommen in Frage, um einen Abschnitt unterzubringen? Alle *cache lines*, die auf diese Weise zusammengehören, werden einem Set zugeordnet.
number_of_sets	Zahl der Sets (s.o.)
size	Produkt aus *coherency_line_size*, *ways_of_associativity* und *number_of_sets*.

Ziel ist es, die zur Verfügung stehenden CPUs auszulasten, d.h. es sollte möglichst wenig Zeit damit verbracht werden, auf Speicherzugriffe zu warten. Dazu gibt es folgende Ansätze:

- Cache-optimierte Datenstrukturen mit möglichst wenig Indirektionen durch Zeiger. Eine Matrix sollte beispielsweise zusammenhängend im Speicher abgelegt werden und nicht mit Hilfe einer Zeigerliste (also beispielsweise mit **double**$**$) realisiert werden.
- Entsprechend kann es sich lohnen, lineare Listen blockweise zu implementieren oder B-Bäume bzw. B*-Bäume statt binären ausgeglichenen Bäumen zu verwenden.
- Einsatz fortgeschrittener Optimierungstechniken wie *instruction schedulung*, ggf. in Verbindung mit *loop unrolling*.

- Diese Technik bemüht sich darum, die Instruktionen (soweit dies entsprechend der Datenflussanalyse möglich ist) so anzuordnen, dass in der Prozessor-Pipeline keine Stockungen auftreten.
- Das lässt sich nur in Abhängigkeit des konkret verwendeten Prozessors optimieren, da nicht selten verschiedene Prozessoren der gleichen Architektur mit unterschiedlichen Pipelines arbeiten.
- Ein recht großer Gewinn wird erzielt, wenn ein vom Speicher geladener Wert erst sehr viel später genutzt wird.
- Beispiel: `x = a[i] + 5; y = b[i] + 3;`
  Hier ist es sinnvoll, zuerst die Ladebefehle für `a[i]` und `b[i]` zu generieren und erst danach die beiden Additionen durchzuführen und am Ende die beiden Zuweisungen.

axpy.c

```
// y = y + alpha * x
void axpy(int n, double alpha, const double* x, double* y) {
   for (int i = 0; i < n; ++i) {
      y[i] += alpha * x[i];
   }
}
```

- Dies ist eine kleine Blattfunktion, die eine Vektoraddition umsetzt. Die Länge der beiden Vektoren ist durch n gegeben, x und y zeigen auf die beiden Vektoren.
- Aufrufkonvention:

Variable	Register
n	%o0
alpha	%o1 und %o2
x	%o3
y	%o4

Resultat von gcc -O3

axpy.s

```
        add     %sp, -120, %sp
        cmp     %o0, 0
        st      %o1, [%sp+96]
        st      %o2, [%sp+100]
        ble     .LL5
        ldd     [%sp+96], %f12
        mov     0, %g2
        mov     0, %g1
.LL4:
        ldd     [%g1+%o3], %f10
        ldd     [%g1+%o4], %f8
        add     %g2, 1, %g2
        fmuld   %f12, %f10, %f10
        cmp     %o0, %g2
        faddd   %f8, %f10, %f8
        std     %f8, [%g1+%o4]
        bne     .LL4
        add     %g1, 8, %g1
.LL5:
        jmp     %o7+8
        sub     %sp, -120, %sp
```

- Ein *loop unrolling* fand hier nicht statt, wohl aber ein *instruction scheduling*.

- Der C-Compiler von Sun generiert für die gleiche Funktion 241 Instruktionen (im Vergleich zu den 19 Instruktionen beim gcc).
- Der innere Schleifenkern mit 81 Instruktionen behandelt 8 Iterationen gleichzeitig. Das orientiert sich exakt an der Größe der *cache lines* der Architektur: $8 * \textbf{sizeof}(\textbf{double}) == 64$.
- Mit Hilfe der prefetch-Instruktion wird dabei jeweils noch zusätzlich dem Cache der Hinweis gegeben, die jeweils nächsten 8 Werte bei x und y zu laden.
- Der Code ist deswegen so umfangreich, weil
  - die Randfälle berücksichtigt werden müssen, wenn n nicht durch 8 teilbar ist und
  - die Vorbereitung recht umfangreich ist, da der Schleifenkern von zahlreichen bereits geladenen Registern ausgeht.

- Der gcc kann mit der Option „-funroll-loops“ ebenfalls dazu überredet werden, Schleifen zu expandieren.
- Bei diesem Beispiel werden dann ebenfalls 8 Iterationen gleichzeitig behandelt.
- Der innere Schleifenkern besteht beim gcc nur aus 51 Instruktionen – ein *prefetch* entfällt und das Laden aus dem Speicher wird nicht an den Schleifenanfang vorgezogen. Entsprechend wird hier das Optimierungspotential noch nicht ausgereizt.

- Hierbei wird auf den Aufruf einer Funktion verzichtet. Stattdessen wird der Inhalt der Funktion genau dort expandiert, wo sie aufgerufen wird.
- Das läuft so ähnlich ab wie bei der Verwendung von Makros, nur gelten weiterhin die bekannten Regeln.
- Das kann jedoch nur gelingen, wenn der Programmtext der aufzurufenden Funktion bekannt ist.
- Das klappt bei Funktionen, die in der gleichen Übersetzungseinheit enthalten sind und in C++ bei Templates, bei denen der benötigte Programmtext in der entsprechenden Headerdatei zur Verfügung steht.
- Letzteres wird in C++ intensiv (etwa bei der STL oder der *iostreams*-Bibliothek) genutzt, was teilweise erhebliche Übersetzungszeiten mit sich bringt.