Institut für Parallele und Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Fachstudie Nr. 198

Vergleich unterschiedlicher Compiler am Beispiel von SG++

Adam Grahovac, Fabian Toth, Patrick Wickenhäuser

Studiengang:	Softwaretechnik
Prüfer/in:	Jun.-Prof. Dr. Dirk Pflüger
Betreuer/in:	Dipl.-Inf. David Pfander
Beginn am:	22. April 2014
Beendet am:	22. Oktober 2014
CR-Nummer:	D.2.8, D.3.4, D4.8

Kurzfassung

In größeren Softwareentwicklungsprojekten ermöglichen Prozessoptimierungen, zum Beispiel durch besseres Tooling, eine erhöhte Effizienz. Der oft eingesetzte Compiler GCC ist ein zentrales Werkzeug in der Softwareentwicklung und das Finden einer besseren Alternative hätte eine große Kostenersparnis zur Folge, der den Aufwand der Prozessoptimierung schnell amortisieren würde.

Diese Fachstudie befasst sich mit den unbekannten Auswirkungen bei der Migration zu Clang am Beispiel der Bibliothek SG^{++}. Betrachtet werden die Übersetzungszeit und Auswirkungen auf die Performance der entstandenen Anwendungen, sowie die Unterstützung benötigter, moderner Features. Um die Compiler in der Praxis zu vergleichen, wurde auf zehn unterschiedlichen Systemen getestet. Die Übersetzungszeit konnte im Schnitt auf ein Drittel reduziert werden, ohne dass es auf die Ausführungszeit der Anwendungen signifikanten Einfluss genommen hat.

Am Ende stand fest, dass GCC durch Clang leicht zu ersetzen ist, wenn man nicht auf einem Windows-System entwickelt oder OpenMP benötigt. Doch überzeugt haben vor allem fortschrittliche Tools und eine erhöhte Benutzerfreundlichkeit. Dadurch kann eine Kostenersparnis resultieren, die den Aufwand der Migration bei Weitem übertrifft.

Abstract

Big software development projects offer process optimizations, e. g. by improved tooling, that result into higher efficiency at project tasks. The GCC Compiler is a widespread tool that represents a central tool in software development, but if it is substituted by a better alternative, high savings of expenses would be the result.

This case study engages with unknown effects when migrating Clang for SG^{++}, focusing on compile speed, runtime performance and support of necessary, modern features. In order to compare the compilers in practice, compile speed and runtime performance were tested on ten different systems. It turns out, that compile speed could be reduced to a third without negative effects on the runtime performance.

In Conclusion, GCC is easily exchangeable by Clang, if not using on a Windows machine or when using OpenMP. Especially the advanced tools and ease of use was very convincing. This results into savings of expenses that compensate the costs of migration easily.

Inhaltsverzeichnis

1 Einleitung

Die Programmbibliothek *SG*$^{++}$ ermöglicht durch die Verwendung von dünnen Gittern [PPB10, G^{+}06] Berechnungen in verschiedenen Anwendungsfällen. Zu diesen gehören zum Beispiel Data Mining und das Lösen von Differentialgleichungen. Um diese Bibliothek benutzen zu können, kann sie mit verschiedenen Compilern übersetzt werden.

SG^{++} unterstützt insbesondere GCC aus der GNU Compiler Collection [gnu] und den C++ Compiler von Intel [icc]. Dabei braucht jeder Compiler unterschiedlich lange für die Übersetzung und generiert unterschiedlich performanten Programmcode. Der Compiler von Intel liefert vor allem für deren Prozessoren die besten Ergebnisse, jedoch ist dessen Nutzung mit hohen Lizenzkosten verbunden. Seit ein paar Jahren wird der neuere Compiler Clang unter einer Opensource Lizenz entwickelt, der dem GCC Konkurrenz machen möchte.

In dieser Fachstudie wird untersucht, ob Clang als Alternative zu GCC für *SG*$^{++}$ eingesetzt werden kann. Dazu wird untersucht, wie sich die Performanz einiger Algorithmen, die die Bibliothek unterstützt, verhält. Außerdem werden die weiteren Vorzüge, wie die besseren Fehlermeldungen und die kürzere Übersetzungszeit untersucht. Dazu werden Beispiele erstellt, die mit *SG*$^{++}$ ausgeführt werden, um dies zu überprüfen. Des weiteren wird in dieser Arbeit untersucht, welche Profilingtools mit Clang oder mit beiden Compilern benutzbar sind.

1.1 Gliederung

Diese Arbeit ist in folgender Weise gegliedert:

Kapitel 2 – Grundlagen beschreibt den fachlichen Hintergrund der Arbeit

Kapitel 3 – Auswirkungen auf Softwareentwicklungsprozesse vergleicht die beiden Compiler in der Theorie

Kapitel 4 – Profiling Tools beschreibt die Auswahl eines geeigneten Profilingtools

Kapitel 5 – Compile Performance vergleicht die Messergebnisse der Kompilierung mit beiden Compilern

Kapitel 6 – Performance von Anwendung vergleicht die Messergebnisse bei der Ausführung einiger Berechnungen mit SG^{++}

Kapitel 7 – Schlussfolgerungen fasst die Ergebnisse der Arbeit nochmals zusammen

2 Grundlagen

In diesem Kapitel werden die Grundlagen der untersuchten Programmbibliothek *SG*$^{++}$ und der verwendeten Compiler erläutert.

2.1 SG++

In vielen numerischen Aufgabenbereichen stößt man auf zu approximierende Funktionen, deren Charakteristik auf verschiedenen Punkten stark variiert. Dadurch wird bei der Verwendung von äquidistanten Gittern entweder an manchen Punkten zu exakt approximiert, was zu einer verhältnismäßig schlechten Laufzeit führt, oder, wenn die Maschenbreite größer gewählt wird, werden wichtige Charakteristika ausgebügelt. Um die Vorteile aus beiden Szenarien mitzunehmen, können anpassungsfähige Gitter verwendet werden. Diese werden an wichtigen Punkten der Funktion enger, damit wichtige Informationen über die Funktion an diesen Stellen mitgenommen werden können. *SG*$^{++}$ ist eine Programmbibliothek, die die Verwendung von anpassungsfähigen Gittern bei der Durchführung zahlreicher Operationen anbietet, z. B. Interpolation und Data-Mining. [PPB10]

2.1.1 Dünne Gitter

Dünne Gitter werden genutzt um Funktionen zu approximieren. Sie basieren auf vollen Gittern, die Basisfunktionen mit kompakten Trägern in hierarchisch aufgeteilten Unterräumen nutzen. Die Aufteilung des Raumes in Unterräume basiert auf einer binären Aufteilung. Die zu approximierende Funktion wird als gewichtete Summe der Basisfunktionen dargestellt. Dünne Gitter unterscheiden sich von vollen Gittern, indem sie nur die Basisfunktionen aus Unterräumen selektieren, die am meisten zur Lösung beitragen. Im Gegensatz zu den vollen Gittern, die jede Basisfunktion mit in die Gewichtung einbeziehen. Die Selektion hilft die Auswirkungen der exponentiellen Abhängigkeit der Dimension auf die Anzahl der benötigten Samples zu reduzieren, ohne dabei die Fehlerordnung der vollen Gitter zu verlieren. Die Wahl

der Basisfunktionen ist abhängig von der Anwendung. Verwendet werden Funktionen mit kompaktem Träger, wie beispielsweise Hat Functions, Wavelets oder B-Splines. [PPB10]

2.1.2 Klassifizierung

Klassifizierung ist eine Gruppe von Algorithmen aus dem Bereich überwachtes Lernen, die die Zugehörigkeit von Punkt-Daten zu Klassen approximieren. Die Ausgabe der Klassifizierung ist die Bezeichnung der approximierten Klassen. Neben den zu klassifizierenden Daten benötigt der Algorithmus einen Training-Datensatz, der zur Erlernung eines Modells, zur Einschätzung der unklassifizierten Daten genutzt wird. Der Training-Datensatz besteht aus Punkt-Daten inklusive deren Klasse. Auch dieser Datensatz kann Fehler beinhalten. Nachdem der Klassifizierungsalgorithmus ein Modell für die Klassifizierung erlernt hat, kann er zur Klassifizierung von Punkt-Daten mit unbekannten Klassen angewendet werden. [PPB10]

2.2 Compiler

In dieser Arbeit werden die zwei Compiler Clang und GCC eingesetzt und verglichen. Dieses Unterkapitel beschreibt die Architekturen der beiden und die internen Abläufe zur Übersetzung des Programmcodes.

2.2.1 Clang

Clang wurde 2007 von Apple vorgestellt und unter einer Opensource Lizenz veröffentlicht. Seit 2009 gilt es als stabil und kann produktiv verwendet werden. Clang ist ein Compiler Front-End, welches verschiedene Sprachen der C Familie in LLVM Zwischencode übersetzen kann.

LLVM ist ein Compiler Back-End welches LLVM Zwischencode optimieren kann, um daraus dann Maschinencode zu erzeugen. Der Ablauf um Programmcode in Maschinencode zu übersetzen ist in Abbildung 2.1 dargestellt.

Abbildung 2.1: Ablauf einer Übersetzung mit Clang

Zunächst erzeugt das Clang Front-end aus dem Programmcode den LLVM Zwischencode. Dies wird dann an das LLVM Back-end weitergereicht, welches zunächst zahlreiche Optimierungen durchführt, um dann mit dem Codegenerator den Maschinencode zu erzeugen [Lat08].

In dieser Arbeit wird der Begriff Clang Compiler dazu verwendet, um diesen ganzen Prozess zu beschreiben und es ist nicht nur das Front-end gemeint.

2.2.2 GCC

Im Jahr 1987 wurde der GNU C Compiler zum ersten Mal mit der Version 1.0 veröffentlicht und wird seitdem unter einer Opensource Lizenz weiterentwickelt. 1999 wurde dieser dann um zusätzliche Programmiersprachen erweitert und bekam dadurch den Namen GNU Compiler Collection.

Die GCC liefert das komplette Front-end und Back-end um Programmcode in Maschinencode zu übersetzen. Der genaue Ablauf von diesem Prozess ist in Abbildung 2.2 dargestellt.

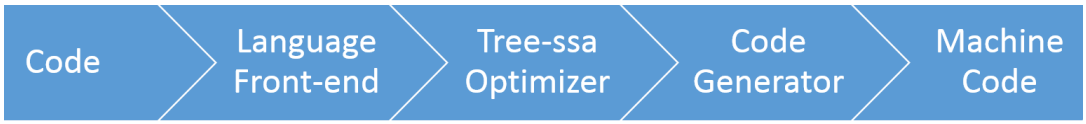

Abbildung 2.2: Ablauf einer Übersetzung mit GCC

Zunächst erzeugt das passende Front-end eine Zwischenpräsentation des Codes und gibt diese an den Optimierer weiter. Dieser erzeugt aus den Syntaxbäumen einen weiteren Zwischencode, der GIMPLE genannt wird. Aus diesem optimierten GIMPLE Code generiert der Codegenerator zunächst noch einen Zwischencode bevor er den Maschinencode erzeugt [Fou].

3 Auswirkungen auf Softwareentwicklungsprozesse

In diesem Kapitel werden einige Auswirkungen der Compiler auf Aspekte der Softwareentwicklung erläutert.

3.1 C++ Feature Support

Für *SG*$^{++}$ ist die Unterstützung von modernen C++ Features und Parallelisierungs-Bibliotheken unverzichtbar. Insbesondere OpenMP und die Umsetzung der C++ 11 Features in der Standardbibliothek sind von Interesse.

3.1.1 OpenMP

Das LLVM Unterprojekt OpenMP beschäftigt sich mit der Umsetzung von OpenMP [clad]. Die Implementierung von OpenMP für GCC wird vom GOMP Projekt umgesetzt. Dadurch unterstützt GCC seit der Version 4.7 OpenMP 3.1 und seit GCC 4.9 OpenMP 4.0. Seit Clang 3.5, das im September 2014 erschien, unterstützt Clang OpenMP. Davor musste spezieller Source Code (clang-omp) [clac] heruntergeladen und selbst kompiliert werden oder auf die Verwendung von OpenMP verzichten werden.

3.1.2 C++ Standardbibliothek

Die unter Linux unterstützen C++ Features der von uns betrachteten Compiler sind der Tabelle 3.1 zu entnehmen. Der aktuelle Status der Implementierung kann für GCC [gcca, gccb] und Clang [claa] in der Online-Dokumentation nachgeschlagen werden.
Unter Windows ist die Clang Standardbibliothek (libc++) momentan nur bedingt einsetzbar, da sie nur wenige moderne Features unterstützt. Als Ersatz kann die GCC Standardbibliothek

Comnpiler	C++11	C++14	C++17
GCC 4.8.1	voll	teilweise	kein
GCC 4.9	voll	teilweise	kein
Clang 3.3	voll	voll	kein
Clang 3.4	voll	voll	experimentell

Tabelle 3.1: C++ Features zweier GCC und Clang Versionen

einer MinGW Installation verwendet werden. Dadurch ist man jedoch auf die von GCC implementierten C++ Features beschränkt.

3.2 Warnungen und Fehlermeldungen

Clang kann als Drop-in-Replacement für GCC genutzt werden und unterstützt alle GCC Compiler-Flags für Warnungen und Fehler. Jedoch verursachen diese Flags bei Clang ein anderes Verhalten als bei der Verwendung derselben mit dem GCC. Bei der Entwicklung von Clang wurde Wert darauf gelegt, die User-Experience im Vergleich zu GCC zu verbessern. Im Allgemeinen kann man sagen, dass Clang bereits ohne Compiler-Flags für Warnungen und Fehler viele Meldungen ausgibt. Ob dieses Verhalten gewünscht ist, muss von Fall zu Fall entschieden werden. Es folgen nun Beispiele, die die oben genannten Punkte verdeutlichen sollen.

3.2.1 Beispiele für den Vergleich von Warnungen

In diesem Abschnitt werden die Warnmeldungen die Clang bzw. GCC ausgeben verglichen.

Array out of Bounds

```
1  #include <iostream>
2
3  int main() {
4      const int tooBigInteger = 13;
5      int smallArray[5];
6
7      int a = smallArray[tooBigInteger];
8      std::cout << a << std::endl;
9  }
```

Abbildung 3.1: smallArray ist an der Stelle 13 nicht definiert.

Das in Abbildung 3.1 dargestellte Array `smallArray` wird in Zeile 7 außerhalb seines Definitionsbereichs ausgewertet. Wie die Compiler darauf reagieren, wird in Abbildung 3.2 dargestellt. Beide Compiler können die Überschreitung des Definitionsbereichs erkennen. Clang findet dies sogar ohne dass man ihm die Option `-Wall` mitgibt. GCC verlangt jedoch noch zusätzlich, dass der Code mit `-O3` optimiert wird.

```
infedyn059:ArrayBounds Adam$ clang++ -c array_boundary_check.cpp
array_boundary_check.cpp:7:10: warning: array index 13 is past the end of the
      array (which contains 5 elements) [-Warray-bounds]
        int a = smallArray[tooBigInteger];
                ^          ~~~~~~~~~~~~~~
array_boundary_check.cpp:5:2:       array 'smallArray' declared here
        int smallArray[5];
        ^
1 warning generated.
infedyn059:ArrayBounds Adam$ g++-mp-4.8 -Wall -O3 array_boundary_check.cpp
array_boundary_check.cpp: In function 'int main()':
array_boundary_check.cpp:7:34: warning: 'smallArray[13]' is used uninitialized i
n this function [-Wuninitialized]
  int a = smallArray[tooBigInteger];
                                  ^
infedyn059:ArrayBounds Adam$ g++-mp-4.8 -Wall array_boundary_check.cpp
infedyn059:ArrayBounds Adam$
```

Abbildung 3.2: Clang erklärt in aller Genauigkeit, was hier nicht stimmt.

Operatorrangfolge

```
int main() {
    int i1 = 0;
    int i2 = 1;

    // Operatorrangfolge?
    bool ret = !i1 == i2;

    return ret;
}
```

Abbildung 3.3: Was bindet stärker? Die Negation oder der Vergleich?

Der Quelltext in Abbildung 3.3 ist für den Compiler eindeutig definiert. Für jemanden, der den Code zum ersten mal liest, aber möglicherweise nicht. Die Negation bindet stärker als der Vergleichsoperator, deswegen wird `ret` im Code auf texttt!(i1) == i2 gesetzt und nicht auf `!(i1 == i2)`. Man kann davon ausgehen, dass der Autor des Quelltextes das so gewollt hat, Clang macht das aber nicht und schlägt in Abbildung 3.4eine Schreibweise vor, die alle Zweifel ausräumt.

```
Adams-MacBook-Air:if_assignment Adam$ g++-mp-4.8 -Wall -c if_assignment.cpp
Adams-MacBook-Air:if_assignment Adam$ clang++ -c if_assignment.cpp
if_assignment.cpp:6:13: warning: logical not is only applied to the left hand
      side of this comparison [-Wlogical-not-parentheses]
        bool ret = !i1 == i2;
                  ^   ~~
if_assignment.cpp:6:13:       add parentheses after the '!' to evaluate the
      comparison first
        bool ret = !i1 == i2;
                  ^
                   (       )
if_assignment.cpp:6:13:       add parentheses around left hand side expression
      to silence this warning
        bool ret = !i1 == i2;
                  ^
                   (  )
1 warning generated.
Adams-MacBook-Air:if_assignment Adam$
```

Abbildung 3.4: Clang warnt vor einem hohen Fehlerpotential.

Uninitialized Member

```
1 class A {
2   int x;
3   int y;
4   A() : x(y) {}
5 };
6
```

Abbildung 3.5: Im Initializer wird x auf y gesetzt, obwohl y noch nicht gesetzt wurde.

In Zeile 4 der Abbildung 3.5 wird ein Konstruktor definiert, der beim Aufruf die Variable x auf den Wert von y setzt, obwohl dieser noch nicht definiert worden sein konnte. Auch hier warnt Clang vor unerwünschten Ergebnissen bei der Codeausführung. GCC bleibt im selben Szenario trotz `-Wall` stumm, siehe Abbildung 3.6.

```
Adams-MacBook-Air:unitialized_member Adam$ clang++ -c unitialized_member.cpp
unitialized_member.cpp:4:11: warning: field 'y' is uninitialized when used here [-Wuninitialized]
  A() : x(y) {}
          ^
1 warning generated.
Adams-MacBook-Air:unitialized_member Adam$ g++-mp-4.8 -Wall -c unitialized_member.cpp
Adams-MacBook-Air:unitialized_member Adam$
```

Abbildung 3.6: GCC ist mit dem Quelltext einverstanden.

3.2.2 Beispiele für den Vergleich von Fehlermeldungen

In diesem Abschnitt werden die Fehlermeldungen die Clang bzw. GCC ausgeben verglichen.

Semikolon

```
1 class Test { int t; }
2 Test instance();
3
```

Abbildung 3.7: Fehlendes Semikolon am Ende der ersten Zeile.

Im Beispiel von Abbildung 3.7 wird eine Klasse definiert, die Definition wird aber nicht mit einem Semikolon abgeschlossen. Abbildung 3.8 zeigt, dass Clang, im Gegensatz zum GCC, die Wurzel des Problems eindeutig identifizieren kann.

```
Adams-MacBook-Air:Semicolon Adam$ clang -c semicolon.cpp
semicolon.cpp:1:22: error: expected ';' after class
class Test { int t; }
                     ^
1 error generated.
Adams-MacBook-Air:Semicolon Adam$ g++-mp-4.8 -c semicolon.cpp
semicolon.cpp:2:6: error: expected initializer before 'instance'
 Test instance();
      ^
Adams-MacBook-Air:Semicolon Adam$
```

Abbildung 3.8: GCC erkennt einen Fehler, aber nicht seine Ursache.

Template-Expansion

```
1  template <typename T>
2  int Y(T x) {
3      return x * x;
4  }
5
6  template <typename T>
7  int X(T x) {
8      return Y(x);
9  }
10
11 int main() {
12     int a = X("gsdg");
13 }
```

Abbildung 3.9: Strings lassen sich nicht multiplizieren.

Die Verwendung von Templates kann Compiler-Errors für den Benutzer schnell undurchsichtig machen, da sich die eigentliche Fehlerursache hinter einer langen Kette von Templates verstecken kann. Dass im Quelltext der Abbildung 3.9 versucht wird, Strings zu multiplizieren, ist auf den ersten Blick nicht ersichtlich. In Abbildung 3.10 sieht man, dass beide Compiler die kausalen Ketten ausgeben, jedoch machen sie das sehr unterschiedlich.

```
Adams-MacBook-Air:TemplateExpansion Adam$ clang++ -c template_expansion.cpp
template_expansion.cpp:5:11: error: invalid operands to binary expression
      ('const char *' and 'const char *')
        return x * x;
               ~ ^ ~
template_expansion.cpp:10:9:       in instantiation of function template
      specialization 'Y<const char *>' requested here
        return Y(x);
               ^
template_expansion.cpp:14:10:       in instantiation of function template
      specialization 'X<const char *>' requested here
        int a = X("gsdg");
                ^
1 error generated.
Adams-MacBook-Air:TemplateExpansion Adam$ g++-mp-4.8 -c template_expansion.cpp
template_expansion.cpp: In instantiation of 'int Y(T) [with T = const char*]':
template_expansion.cpp:10:12:   required from 'int X(T) [with T = const char*]'
template_expansion.cpp:14:18:   required from here
template_expansion.cpp:5:11: error: invalid operands of types 'const char*' and
 'const char*' to binary 'operator*'
  return x * x;
           ^
Adams-MacBook-Air:TemplateExpansion Adam$
```

Abbildung 3.10: Clang führt den Benutzer an die Ursache des Fehlers heran, in umgedrehter Reihenfolge.

3.2.3 Bewertung der Unterschiede zwischen den Compilern

Die Beispiele zeigen, dass Clang vieles anders macht. Bei dem ersten Versuch *SG*$^{++}$ mit Clang zu kompilieren, wurde einige Fehlermeldungen ausgegeben. Erst die Verwendung der folgenden Flags ermöglichte es, die Bibliothek mit Clang erfolgreich zu übersetzen:

```
-Wno-sign-conversion
-Wno-header-guard
-Wno-overloaded-virtual
-Wno-unused-private-field
-Wno-unneeded-internal-declaration
-Wno-self-assign-field
-Wno-parentheses-equality
```

Damit konnte der Code ohne Fehler kompiliert werden. Clang behandelt potentielle Fehlerquellen im Source-Code also strenger und er lässt dem Benutzer dabei auch keine Wahl, da er Warnungen trotz fehlendem `-Wall` ausgibt. Die oben gezeigten Beispiele zeigen aber auch, dass der jüngere Compiler die Fehlerursachen für den Benutzer besser visualisiert. Dieses Feature wird *Expressive Diagnostics* genannt [clab]. Die Standard-Ausgabe listet die betroffenen Ausschnitte auf und unterstreicht die fehlerhafte Stelle durch die Zeichenfolge

~~~ und die Position durch das Zeichen ^. Neben der Fehlermeldung werden Ausschnitte des betroffenen Codes ausgegeben.

So kann es von Fall zu Fall möglich sein, dass im Vergleich zum GCC mehr Fehler gefunden werden und diese schneller behoben werden können.

3.3 Toolchain

In diesem Unterkapitel werden die Tools vorgestellt, die von beiden Compilern mitgeliefert werden.

3.3.1 Tools von Clang

Clang bietet einige Tools, zu denen der GCC keine vergleichbaren Programme anbietet. Diese werden hier vorgestellt.

LibTooling und Clang Check

Die LibTooling Bibliothek von Clang ist eine Bibliothek, mit der eigenständige Programme auf Basis des Front-Ends von Clang entwickelt werden können. Damit können zum Beispiel Syntaxanalysen des Codes durchgeführt werden. Einige der folgenden Tools wurden mit dieser Bibliothek entwickelt [too]

ClangCheck ist eine Bibliothek, die die LibTooling Bibliothek enthält, aber eher auf Fehleranalysen mit Hilfe der Basis von Clang spezialisiert ist. Mit ihr können Fehlerprüfungen durchgeführt werden und es kann direkt auf den abstrakten Syntaxbaum zugegriffen werden [clae].

Die Gnu Compiler Collection bietet keine Möglichkeit, Tools auf Basis des Compilers zu bauen. Dadurch können damit keine eigenständige Tools entwickelt werden.
~~~

Clang Format

Clang Format kann vorhandenen Code nach bestimmten Richtlinien formatieren. Diese können eigenhändig konfiguriert werden, oder es können die Vorlagen, welche sich an bekannten Styleguides orientieren und von dem Programm mitgeliefert werden, benutzt werden. Das Tool benutzt für die Formatierung den Lexer von Clang und generiert aus dem Quelltext einen Token Stream. In diesem verändert es die vorhandenen Whitespaces und generiert daraus wieder Code der den Formatierungsregeln entspricht. Dieses Tool gibt es eigenständig und als Teil des Clang Compilers, um es auch von Entwicklungsumgebungen ansteuern zu können. Die eigenständige Version kann mit dem Befehl `clang-format [Optionen] [<Quelldatei>]` aufgerufen werden [for].

Clang Modernize

Clang Modernize ist ein eigenständiges Tool, welches C++ Code, der alten Standards entspricht in Code konvertiert der Konstrukte des neuesten Standards benutzt. Es gibt mehrere Risikolevel für die unterschiedlichen Konvertierungen. Bei einem höheren Risikolevel steigt die Chance, dass das Programm fehlerhaften Code erzeugt.

Die folgenden Codekonstrukte können verarbeitet werden [modb]:

- Konvertierung von normalen For-Schleifen zu Ranged-based-For-Schleifen
- Konvertiert eine Nullpointer Konstante (NULL, 0) zu nullptr
- Konvertiert spezifischen Typspezifizierer zu auto-Typspezifizierer
- Fügt override zu Methoden hinzu, die eine virtuelle Methode überschreiben
- Ersetzt std::auto_ptr mit std::unique_ptr

Clang Modernize kann als eigenständiges Programm über die Kommandozeile aufgerufen werden. Dazu wird der Befehl `clang-modernize [Optionen] [<Quelldatei>]` benutzt [moda].

Clang Static Analyzer

Der Clang Static Analyzer ist ein Programm, um statische Codeanalysen während der Kompilierung durchzuführen. Dafür arbeitet er während der Kompilierung direkt mit dem Zwischencode von LLVM, wodurch die Übersetzung bis zu drei Mal so lange dauern kann. Dabei können zum Beispiel Dinge, wie Code der niemals ausgeführt wird, oder Divisionen mit 0 gefunden werden. Die statische Analyse kann mit dem Compilerflag `scan-build make <Parameter>` aufgerufen werden [sta].

Der Clang Static Analyzer bietet zudem eine Schnittstelle zur Benutzung durch Entwicklungsumgebungen, die schon in Apples XCode System genutzt wird. Es befindet sich momentan noch im Betastadium, was vor allem durch den Anteil an falsch-positiven Meldungen bemerkbar ist. An diesem Problem wird aber aktiv gearbeitet.

MemorySanitizer (MSan)

Der Memory Sanitizer ist seit Version 3.3 in Clang integriert, befindet sich aber trotzdem in einem experimentellen Zustand und hat noch einige Fehler. Der Memory Sanitizer kann uninitialisierte Leseoperationen erkennen. Er besteht aus einer Instrumentationskomponente, die den Code während der Kompilierung instrumentiert und einer Laufzeitkomponente, die den Code zur Laufzeit auswertet. Die Ausführungszeit und der Speicherverbrauch können sich zur Laufzeit bis zu verdreifachen. Die Analyse mit dem Memory Sanitizer kann mit dem Compilerflag `-fsanitize=memory` aktiviert werden [mem].

3.3.2 Tools beider Compiler

In diesem Unterkapitel werden die beiden Tools vorgestellt, die zunächst nur für Clang entwickelt wurden, dann aber auch in GCC integriert wurden.

AddressSanitizer (ASan)

Der AddressSanitizer wurde von Google entwickelt und bietet die Möglichkeit den Code während der Übersetzungszeit zu instrumentieren, um dann zur Laufzeit Adressfehler bei dem Zugriff auf den Hauptspeicher zu finden. Es werden *out-of-bounds* Zugriffe auf den *Heap*, den *Stack* und auf globale Objekte erkannt. Außerdem werden *use-after-return* Fehler

gefunden. Der Address Sanitizer kann mit dem Compilerflag *-fsanitize=address* eingeschaltet werden.

Der AddressSanitizer ist seit Version 3.1 in Clang und seit Version 4.8 in GCC integriert. Bei der Verwendung wird der Speicherverbrauch bis zu drei mal so groß und die Ausführungszeit des Programmes kann sich bis zu verdoppeln [SBPV12].

ThreadSanitizer (TSan)

Der ThreadSanitizer wurde von Google entwickelt und bietet die Möglichkeit, den Code während der Übersetzungszeit zu instrumentieren, um dann zur Laufzeit Fehler beim Threadmanagement zu finden. Es können *Data Races*, *Deadlocks*, *unjoined Threads* und die Zerstörung von gesperrten Mutexen gefunden werden. Der Sanitizer kann mit dem Compilerflag `-fsanitize=thread` aktiviert werden.

Der ThreadSanitizer ist seit Version 3.2 in Clang und seit Version 4.8 in GCC integriert. Bei der Verwendung kann der Speicherverbrauch bis zum Faktor zehn größer werden und die Ausführung kann bis zu 20 Mal so lange dauern. Das Tool befindet sich momentan noch im Betastatus und wird stetig weiterentwickelt [SI09].

3.3.3 Vergleich der Toolchain

GCC bietet außer den beiden Tools die auch im Clang integriert sind keine Programme die zu den restlichen der Toolchain von Clang vergleichbar wären. Da es in dieser Arbeit ausschließlich um den Vergleich der beiden Compiler geht, werden die Programme, die üblicherweise mit dem GCC verwendet werden, aber nicht direkt zur Gnu Compiler Collection gehören, nicht verglichen.

4 Profiling Tools

In diesem Abschnitt wird erläutert, wobei es sich um Profiling handelt. Anschließend werden auf einige Profiling-Ansätze, Tools und Techniken eingegangen und jeweils ihre Nützlichkeit in Bezug auf den Vergleich der Compiler gegenübergestellt und das passende Tool ausgewählt.

4.1 Profiling

Beim Profiling werden Informationen über die Ausführung einer Software gesammelt, um Aussagen über den zeitlichen Verlauf und die Ressourcennutzung machen zu können. Diese Informationen werden häufig zur Identifikation von problematischen Routinen und zur Optimierung genutzt [CMH91]. Es haben sich zwei Profiling Ansätze bewährt: Statistische Profiling Verfahren und Instrumentierung, die in den folgenden Abschnitten erläutert werden. Diese Verfahren bieten unterschiedliche Vor- und Nachteile, sowie unterschiedliche Aussagen über den Verlauf der Ausführung einer Anwendung.

Neben dem zeitlichen Verlauf der Anwendung sind weitere Attribute der Anwendung von Interesse. Dazu zählen unter anderem das Caching-Verhalten und Branch Prediction. Diese Attribute lassen sich mit Hilfe der Performance Counter der Performance Monitoring Unit (PMU) der CPU messen. Performance Counter sind Hardware Register, die einige Ereignisse zählen, darunter auch Cache-Zugriffe und Branch Predictions. Ebenfalls von Interesse ist das Verhalten des Profilers im Kernel Mode, daher das Verhalten während der Wartezeit auf Ressourcen.

4.1.1 Statistisches Profiling

Beim statistischem Profiling sind keine Änderungen zur Compilezeit notwendig. Die Messungen werden in regelmäßigen Intervallen durch Hardware-Interrupts vorgenommen, bei denen die aktuelle Position im Code ermittelt und mit Zusatzinformationen gespeichert wird.

Dieser Satz an Informationen wird als “Sample“ bezeichnet. Aus der Menge an Samples können statistische Aussagen über den relativen Bearbeitungsaufwand einzelner Instruktionen gemacht werden. Sie beinhalten aber im Normalfall keine absoluten Zeitpunkte. Abhängig von der Frequenz der Interrupts und damit Anzahl der Samples wird die Ausführung stark verlangsamt. Dadurch wird zwar die Gesamtausführungszeit beeinflusst, die relativen Größen bleiben jedoch erhalten. Debug Symbole werden vom Profiler verwendet um die Abbildung von Maschinen-Instruktionen zu den jeweiligen Funktionen herzuleiten.

Es gibt Compiler-Optimierungen, wie Inlining, welche die Interpretation der Ergebnisse erschweren können. Im Falle von Inlining werden Funktionsaufrufe durch eingesetzte Codesegmente ersetzt. Das führt dazu, dass das Sample nicht in der eigentlichen optimierten Funktion erscheint, sondern in der aufrufenden Funktion. Daher empfiehlt es sich die Anwendungen, vor der Messung, mit Debug Symbolen und ohne Inlining zu kompilieren (*-g -fno-inline*). Zusätzlich können Informationen der Performance Counter der Performance Monitoring Unit (PMU) mit ausgelesen werden. Häufig wird die Anzahl der Last Level Cache (LLC) Zugriffe und Misses ausgelesen.

4.1.2 Instrumentierung

Bei der Instrumentierung wird Messungs-Code meistens zur Compilezeit in den Code eingefügt. Dabei handelt es sich meist um Informationen über die aktuelle Position im Code und einem Zeitstempel. Einige automatische Instrumentierungen fügen Code beim Betreten und Verlassen von Funktionen ein. Es gibt auch Profiler, die vom Entwickler verlangen Messungs-Code manuell einzusetzen.

Diese Art Messungen vorzunehmen bringt in den meisten Fällen nur wenig Overhead mit sich. Instrumentierung hat daher, verglichen mit Sampling, nur geringe Auswirkung auf die Gesamtausführungszeit.

4.1.3 Vergleich

Instrumentierung hat zwar einen deutlich geringeren Overhead, jedoch ist die längere Ausführungszeit, die statistische Profiler Verursachen, für unsere Anforderung akzeptabel, solange die Ergebnisse auch über kleinere Eingabegrößen hinweg konsistent sind und plausible Ergebnisse liefern. Neben der Erkennung von Hotspots und Zeitmessungen sollen auch andere Auswirkungen auf das Kompilat sichtbar sein. Unter anderem Branch-Prediction und das Caching-Verhalten unter Verwendung von PMU-Events.

Durch die relativen Zeiten der statistischen Profiler spiegelt sich die Gesamtausführungszeit nicht in den Ergebnissen wieder, wodurch der direkte Vergleich zwischen verschiedener Hardware ermöglicht wird. Von Instrumentierung vorausgesetzte Modifikationen am Buildprozess könnten unerwartete Seiteneffekte mit sich bringen, welche die Vergleichbarkeit der Messungen gefährden. Ebenso ist die Auswirkung von Instrumentierungscode auf das Caching-Verhalten unklar. Aus diesen Gründen haben wir uns dazu entschieden ausschließlich statistische Profiler zu verwenden.

4.2 Profiling Tools

Unter Verwendung von Linux kommen viele Tools in Frage, von denen wir einige ausgewählt haben und auf diese in diesem Abschnitt einzeln eingehen, um im Anschluss das für unsere Zwecke am besten geeignete Tool auszuwählen.

4.2.1 Perf

Ein wichtiges Tool ist das im Linux Kernel enthaltene Tool Perf. Es ermöglicht Sampling durch den Befehl *perf record*. Die Frequenz des Samplings kann, mit der Option *-F*, in Hertz angegeben werden. Die Messungen können durch Perf in Textform mit Hilfe von *perf report* ausgegeben oder mit einer anderen Messung mit *perf diff* gegenübergestellt werden [per]. Auch im Fehlerfall wird die Ausgabe erzeugt, weil die Messungen direkt vom Linux Kernel gesteuert werden. Dadurch werden auch Messungen im Kernel Mode durchgeführt, während die Anwendung blockiert ist, weil sie beispielsweise auf eine Ressource wartet.

Neben Sampling ermöglicht Perf auch Zugriffe auf die Performance Counter der PMU-Hardware durch den Befehl *perf stat*. Einige der aufgezeichneten Events sind:

- cache-misses
- cache-references
- branch-misses
- branches
- instructions
- cycles

- page-faults
- CPU-migrations
- context-switches
- task-clock-msecs
- seconds time elapsed

Um die exakte Bedeutung der Events zu verstehen wird die Beschreibung der Hardware Counter der CPU benötigt. Diese können in der Dokumentation der CPU nachgeschlagen werden. Das Mapping der Events auf die Hardware Counter hat sich als komplex erwiesen. Dazu haben wir die Tabellen der Events im Linux Kernel [ker, Z. 2185ff.] verwendet, die es ermöglichen den Namen des Events auf den Hardware Performance Counter durch seinen Hex-Code [ker, Z. 22ff.] zurückzuführen, aus diesem wiederum die Beschreibung der CPU Dokumentation bezogen werden kann. Daraus konnten wir die genau Bedeutung der Events ableiten. Daraus ergibt sich beispielsweise, dass es sich beim Event “cache misses“ um die Last Level Cache Misses handelt.

4.2.2 Valgrind

Valgrind ist ein Framework für Softwareanalysetools wie Callgrind, Cachegrind, Memcheck oder Nullgrind und bietet ein GUI durch KCachegrind. Es wurde entwickelt, um keine Änderungen an der Software, durch Linking von Libraries, Preloading, Änderungen am Source Code oder ähnlichem vorauszusetzen [valb]. Stattdessen simuliert es die Ausführung der Anwendung auf einer virtuellen Maschine, mit dessen Hilfe detaillierte Informationen über die Ausführung gesammelt werden können. Valgrind ist derzeitig nicht unter Windows verwendbar.

Für unsere Messungen ist das Tool Callgrind geeignet, das beim Profiling relative Zeitmessungen inklusive Informationen über das Caching-Verhalten sammelt. Debug-Informationen der Anwendung und ihrer Bibliotheken werden verwendet, um die Verbindung zum Source Code herzustellen. Bei der Ausführung muss man mit großem Performanceverlust rechnen. Die Ausführungszeit wächst auf ungefähr das 10 bis 50 fache an. Dies ist auf die Simulation der CPU zurückzuführen.

Die Auswirkungen der virtuellen Maschine auf Branch Prediction, Speicherallokationen, dem Caching-Verhalten und weitere performancekritische Eigenschaften der Software sind

dabei unklar und nur schwer aus der vorhandenen Dokumentation herzuleiten. Aus dieser kann beispielsweise abgeleitet werden, dass der Branch Predictor eines Desktop-Prozessors aus dem Jahre 2004 simuliert wird, der vermutlich große Unterschiede zu den aktuellen Prozessoren in unseren Messungen aufweist. [vala] Daraus könnten unerwartete Verzerrungen der Messergebnisse folgen.

4.2.3 Google Performance Tools

Google Performance Tools ist ein Softwarepaket von Google. Es beinhaltet die folgenden Tools: TCMalloc, einer performanten Implementierung von malloc, einem Heap Profiler, einem Heap Checker und einem CPU Profiler. Für uns ist nur der CPU Profiler von Bedeutung [gooa]. Um den CPU Profiler verwenden zu können muss *-lprofile* gelinked oder preloaded werden. Modifikationen an der Kompilierung müssen nicht vorgenommen werden. Der Pfad zur Ausgabedatei und die zu messende Binärdatei wird über die Umgebungsvariable "CPUPROFILE" spezifiziert. Auf die selbe Weise kann die Frequenz der Interrupts in Hertz durch die Umgebungsvariable "CPUPROFILE_FREQUENCY" bestimmt werden. [goob].

Der CPU Profiler führt Sampling durch, erlaubt aber keinen Zugriff auf Performance Counter. Die Ausgabedatei des Profilers kann mit dem Tool *pprof* analysiert werden, oder in das Callgrind-Format konvertiert und anschließend mit der Valgrind GUI "KCachegrind" analysiert werden.

4.2.4 gprof

Gprof ist ein seit 1982 veröffentlichtes Profiling Tool. Es ist ein Hybrid aus Instrumentierung und Sampling, daher sind Modifikationen zur Compilezeit notwendig, welche sich bei GCC auf eine zusätzliche Compileroption *-pg* beschränkt, die für *Performance Graph* steht und Instrumentierungscode einfügt [GKM82]. Eine Implementierung der *-pg* Option ist für Clang derzeitig weder vorhanden, noch in naher Zukunft abzusehen. Bei der Messung muss zwischen dem *flat profile* und dem *call graph* unterschieden werden. Beim *flat profile* handelt es sich um Messungen der Aufenthaltszeiten des Programms in jeder Funktion, sowie dessen Anzahl der Aufrufe. Der *call graph* zeichnet alle Aufrufe jeder Funktion und deren Aufrufe anderer Funktionen, inklusive der Anzahl, auf. Zusätzlich befinden sich darin Schätzungen über die Aufenthaltswahrscheinlichkeiten der Unterfunktionen [gpra, GKM82, gprb].

Solange sich die Anwendung im Kernel Mode befindet, finden aber keine Messungen statt. Bei der Ausführung schreibt die instrumentierte Anwendung, kurz vor ihrem Ende, eine

Datei namens *gmon.out* in das Ausführungsverzeichnis. Die Ausgabedatei wird im Fehlerfall nicht geschrieben, wodurch die Messungen der Ausführung verloren gehen. Diese Datei kann mit dem Kommandozeilen-Programm Gprof in Textform analysiert werden oder mit Hilfe von *pgprof2dot* ein Bild des Callgraphen erstellt werden. Gprof hat einen höheren Overhead als reine Instrumentierungsprofiler.

4.2.5 Vergleich

Gprof scheidet mangels Clang-Kompatibilität aus. Perf ist äußerst einfach zu installieren und benötigt keine Änderungen des Kompilats, was für die, per Skript ausgeführten, Messungen auf verschiedenen Systemen mit beschränkten Rechten sehr hilfreich ist. Ebenso benötigt Perf keine Modifikationen des Buildprozesses, die mögliche Fehlerquellen sind. Gprof und Google Performance Tools benötigen jeweils Modifikationen zur Compilezeit der SG^{++} Bibliothek, Valgrind und Perf hingegen nicht.

Der Vergleich der Profile ist im Fall von Perf sehr einfach mit Hilfe des Befehls *perf diff* durchführbar. *perf diff* benötigt zwei Profile als Eingabeparameter, deren Messungen tabellarisch gegenübergestellt werden. Dies ist ideal für den Vergleich der Paare von Profilen, der beiden Compiler, geeignet.

Valgrind ermöglicht zwar gute Messungen und Analysen des Caching-Verhaltens, dennoch haben wir uns gegen Valgrind entschieden, da es deutlich langsamer als Perf ist und das genaue Verhalten der virtuellen Maschine nur sehr schwer aus der Dokumentation zu entnehmen ist. Aus diesen Gründen haben wir uns für die Verwendung von Perf entschieden.

5 Compile Performance

Bei der Entwicklung großer Programme, die oft übersetzt werden müssen ist die Übersetzungsdauer ein wesentlicher Faktor. In dieser Fachstudie soll untersucht werden, ob die Annahme, dass Clang schneller kompilieren kann, stimmt.

5.1 Versuchsaufbau

SG++ wurde auf zehn unterschiedlichen Systemen, jeweils mit GCC und Clang, kompiliert und es wurde die Dauer dieses Vorgangs gemessen. Da Clang zum Zeitpunkt der Messungen noch kein OpenMP unterstützt hat, wurden beide Compiler jeweils ohne diese Bibliothek kompiliert, siehe dazu Kapitel 3.1.1.
Verwendete Compilerversionen:

- GCC: 4.8
- Clang: 3.4

Gleichzeitig wurde überprüft welcher der beiden Compiler mit steigender Anzahl an Threads besser skaliert. Dabei lagen Betriebssystem, Compiler und Quellcode immer auf demselben Datenträger. Die Spezifikationen der Testsysteme sind folgende:

5.1.1 Spezifikation der Testsysteme

Id	Fabian	Patrick
CPU	i7-4770k	i7-4700MQ
Cores	4	4
Threads	8	8
RAM	16 GB 1600MHz DDR3	8 GB DDR3 1600 MHz
Mainboard	Asus Z87 Pro	Lenovo VIQY0Y1
HDD/SSD	Samsung 840 256 GB SSD	Liteonit LCS 256 GB SSD

Id	Adam	ISTE_2
CPU	Phenom II X6 1055T	i5-3470
Cores	6	4
Threads	6	4
RAM	4 GB DDR3 1333 MHz	16 GB DDR3 1600 MHz
Mainboard	Gigabyte GA870A-UD3	HP CZC2261WTJ
HDD/SSD	Sharkoon 32 GB USB 2.0	Intel SSD 530 180 GB

Id	ISTE_SSD	ISTE_USB
CPU	i3-2120	i3-2120
Cores	2	2
Threads	4	4
RAM	8 GB DDR3 1333 MHz	8 GB DDR3 1333 MHz
Mainboard	HP CZC2261WTJ	
HDD/SSD	Intel SSD 530 180 GB	Sharkoon 32 GB USB 2.0

Id	SGS	MacBook
CPU	i5-4670	i5-2557M
Cores	4	2
Threads	4	4
RAM	8 GB DDR3 1600MHz	4 GB DDR3 1333 MHz
Mainboard	Fujitsu D3222-A1	MacBook Air Mid 2011
HDD/SSD	Sharkoon 32 GB USB 3.0	Sharkoon 32 GB USB 2.0

Id	VGPU1	Helium
CPU	2x Xeon E5-2620	4x Xeon E7540
Cores	12	24
Threads	24	48
RAM	256 GB	512 GB
Mainboard		
HDD/SSD	HDD	HDD

Die Systeme ISTE_SSD und ISTE_USB sind dieselben Systeme, nur das das eine mal Betriebssystem, Compiler und Quellcode auf der SSD lagen und das andere mal auf einem USB-Stick.

5.2 Ergebnisse

5.2.1 Übersicht aller Ergebnisse

id	CPU	Compilezeit		Zeitgewinn
		GCC	Clang	durch Clang
Adam	Phenom II X6 1055T	121	31	297%
SGS	i5-4670	97	25	284%
Patrick_SSD	i7-4700MQ	108	31	244%
ISTE_USB	i3-2120	201	64	211%
ISTE_SSD	i3-2120	196	64	207%
ISTE2	i5-3470	108	36	198%
Fabian	i7-4770k	99	34	194%
MacBook	i5-2557M	296	107	177%
VGPU1	2x Xeon E5-2620	98	47	108%
Helium	4x Xeon E7540	92	50	82%

in Sekunden

5.2.2 Compilezeiten aller Systeme mit unterschiedlicher Anzahl Threads

Adam

#Threads	1	2	3	4	6	8	12	16
GCC	519271	291906	206551	179750	134048	122984	123141	121396
Clang	161869	82421	55922	47047	31237	30576	30910	31371

Angaben in Millisekunden

Fabian

#Threads	1	2	3	4	6	8	12	16
GCC	300350	170030	131568	113910	104607	99080	99078	102022
Clang	107791	57520	43621	37665	35263	33739	33802	33774

Angaben in Millisekunden

MacBook

#Threads	1	2	3	4	6	8	12	16
GCC	520522	333640	308567	295800	307664	309580	347757	320670
Clang	186167	109798	111116	106636	107364	119173	121776	122140

Angaben in Millisekunden

Patrick_SSD

#Threads	1	2	3	4	6	8	12	16
GCC	307966	169760	130394	126013	112710	109299	108386	108459
Clang	102913	51998	36204	34489	34231	31583	31467	31655

Angaben in Millisekunden

ISTE_SSD

#Threads	1	2	3	4	6	8	12	16
GCC	415581	238794	207550	195718	196642	198205	199986	201093
Clang	149953	76839	69262	64079	63837	64355	64581	65028

Angaben in Millisekunden

ISTE_USB

#Threads	1	2	3	4	6	8	12	16
GCC	445809	265168	232558	206063	202900	202038	200775	201101
Clang	154034	79091	71275	64982	64494	64848	65152	66190

Angaben in Millisekunden

ISTE2

#Threads	1	2	3	4	6	8	12	16
GCC	350158	185548	133669	108142	108495	112388	113085	111829
Clang	137395	66558	46278	36911	36253	36569	36898	37013

Angaben in Millisekunden

SGS

#Threads	1	2	3	4	6	8	12	16
GCC	313435	205816	136119	111234	100544	97449	101535	97518
Clang	95151	48317	33184	27127	25374	25738	26072	26708

Angaben in Millisekunden

VGPU1

#Threads	1	2	3	4	6	8	12
GCC	575377	309421	231355	192785	153449	136010	121669
Clang	252235	132689	100962	83864	68179	61833	56685

#Threads	16	24	32	48	64	96	128
GCC	108250	101226	98506	97964	98345	98177	98791
Clang	51197	47759	47439	47549	47306	47170	47540

Angaben in Millisekunden

Helium

#Threads	1	2	3	4	6	8	12
GCC	854138	451093	285376	242293	182688	148896	126277
Clang	437831	208103	129132	116795	85610	72414	61350

#Threads	16	24	32	48	64	96	128
GCC	111421	104023	97935	92641	91792	92439	91742
Clang	55481	51078	50431	51323	51321	50942	51425

Angaben in Millisekunden

5.2.3 Performancegewinn durch Multithreading

GCC

id	CPU	Compilezeit		Performancegewinn
		1 Thread	Multithreaded	durch Multithreading
Helium	4x Xeon E7540	854	92	831%
VGPU1	2x Xeon E5-2620	575	98	487%
Adam	Phenom II X6 1055T	519	121	328%
ISTE2	i5-3470	350	108	224%
SGS	i5-4670	313	97	222%
Fabian	i7-4770k	300	99	203%
Patrick_SSD	i7-4700MQ	308	108	184%
ISTE_USB	i3-2120	446	201	122%
ISTE_SSD	i3-2120	416	196	112%
MacBook	i5-2557M	521	296	76%

in Sekunden

Clang

id	CPU	Compilezeit		Performancegewinn
		1 Thread	Multithreaded	durch Multithreading
Helium	4x Xeon E7540	438	50	768%
VGPU1	2x Xeon E5-2620	252	47	435%
Adam	Phenom II X6 1055T	162	31	429%
ISTE2	i5-3470	137	36	279%
SGS	i5-4670	95	25	275%
Patrick_SSD	i7-4700MQ	103	31	227%
Fabian	i7-4770k	108	34	219%
ISTE_USB	i3-2120	154	64	139%
ISTE_SSD	i3-2120	150	64	135%
MacBook	i5-2557M	186	107	75%

in Sekunden

5.3 Bewertung

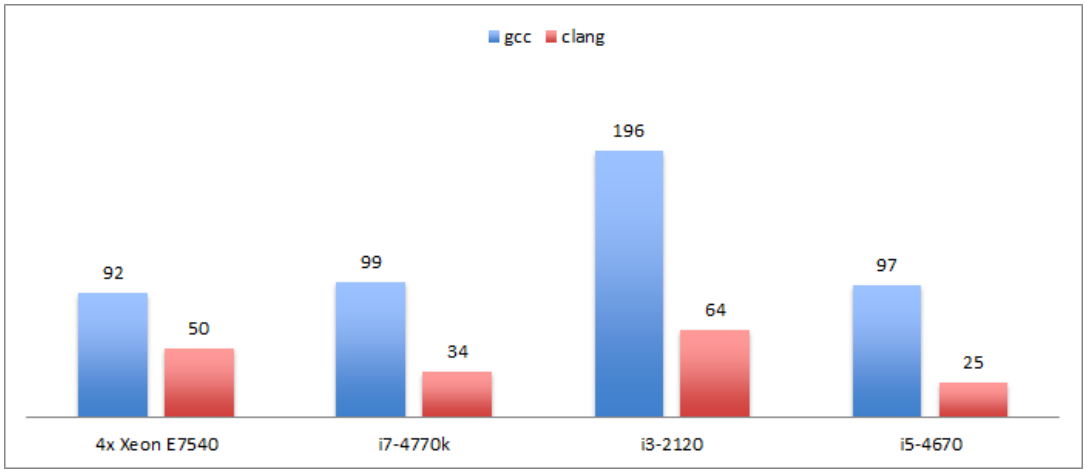

Abbildung 5.1: Veranschaulichung des Performanceunterschiedes

Die Messungen zeigen, dass Clang ausnahmslos schneller kompiliert hat als der GCC. Abbildung 5.1 stellt dar, wie sehr diese Performanceunterschiede vom verwendeten System abhängen. Während auf Multi-Sockel-Systemen der Vorsprung von Clang relativ gering ist, wird dieser bei einem neueren i7-Prozessor und einem etwas älteren i3-Prozessor etwas ausgebaut. Den fast größten Performanceunterschied gelang einem neueren i5-Prozessor, der nur von einem etwas älteren AMD-Sechskerner leicht übertroffen wurde.

Die in Kapitel 5.2.3 gezeigte Gegenüberstellung zeigt, dass der GCC auf Mehrsockelsystemen mit mehreren Threads besser skaliert, auf Einprozessorsystemen aber meistens der Clang. Dass GCC auf manchen Systemen besser skaliert ist aber unerheblich, da der Clang mit einem Thread schneller kompiliert als der GCC mit 16, siehe dazu die Tabellen des Kapitels 5.2.2.

Abschließend lässt sich sagen, dass der Clang wesentlich schneller ist. Diese Tatsache allein kann, bei der Frage eines Umstiegs auf Clang, entscheidend sein.

6 Performance von Anwendung

In diesem Kapitel werden die Auswirkungen der Compiler auf die Ausführungszeiten ausgewertet. Mit Hilfe von Profiling Tools werden einige Anwendungen von SG++ untersucht, indem ihre Performance unter Verwendung der mit GCC und Clang kompilierten SG++ Bibliotheken gegenübergestellt werden. Untersucht werden Anwendungen aus den Bereichen Partielle Differentialgleichungen, Data Mining und Quadratur.

6.1 Versuchsaufbau

Alle Messungen werden für GCC und Clang mit den selben Anwendungen, Daten- und Parametersätzen durchgeführt, um möglichst gleiche Voraussetzungen zu schaffen. Die SG++ Bibliothek wird unmittelbar vor der Ausführung der Test-Anwendungen kompiliert, nachdem bereits vorhandene Kompilate gelöscht wurden. Die Kompilierung der Bibliothek, Anwendungen und Ausführungen werden zentral aus einem Skript gesteuert, um Fehler durch manuelle Bedienung zu vermeiden und die Wiederverwendbarkeit auf verschiedenen Systemen sicherzustellen.

Die Messungen der Ausführungen werden mit *perf record* und *perf stat* durchgeführt. Sobald die Ausführungen mit beiden Compilern gemessen wurden, werden die Ergebnisse mit *perf diff* verglichen. Es wurden sieben Testsysteme zur Messung verwendet, darunter einen AMD Prozessor, wie im vorherigen Kapitel 5.1.1 beschrieben.

6.1.1 Partielle Differentialgleichungen

Zur Messung der Auswirkungen auf die Performance bei der Lösung von partiellen Differentialgleichungen wird der in *SG*$^{++}$ enthaltene Wärmeleitungsgleichung-Löser “HESolver“ und die Finanzmathematik-Anwendung “BlackScholes“ eingesetzt. Die Parametersätze richten sich jeweils an denen im Anwendungs-Code enthaltenen Beispielparametern.

6.1.2 Data Mining

Zur Evaluation der Data Mining Performance wird die Anwendung “ClassifyBenchmark“ auf einen fünfdimensionalen Schachbrett-Datensatz angewendet. Die Anwendung klassifiziert den Datensatz, nachdem sie den dazugehörigen Trainings-Datensatz verarbeitet hat.

6.1.3 Quadratur

Die *SG*$^{++}$ Software beinhaltet ein Beispiel zur Verwendung von *SG*$^{++}$ für numerische Quadratur “c++_simple_quadrature“, das zur Messung verwendet wurde. Die Anwendung erstellt zu Beginn ein reguläres Gitter und befüllt dieses mit Funktionsauswertungen an den jeweiligen Stellen, um die Funktion zu diskretisieren. Das entstandene Gitter wird anschließend zu einem hierarchischem Gitter umgeformt. Dieses wird anschließend auf verschiedene Weisen numerisch integriert.

6.2 Ergebnisse

Black Scholes

id	CPU	Ausführungszeit gcc	Ausführungszeit clang	Zeitgewinn durch Clang
Adam	X6 1055T	147	143	3%
Fabian	i7-4770k	54	53	2%
ISTE_2	i5-3470	80	81	-1%
ISTE	i3-2120	121	117	3%
Patrick	i7-4700MQ	90	86	5%
VGPU1	2x Xeon E5-2620	51	53	-4%
SGS	i5-4670	70	72	-3%

in Sekunden

Auffallend sind die überwiegend geringen Unterschiede zwischen den Ausführungszeiten der beiden Compiler. Lediglich der Heat Equation Solver beinhaltet Differenzen von -10% bis 24%. Insgesamt interpretieren wir die Performanceunterschiede als insignifikant und daher ist Clang zur Ersetzung von GCC, unter dem Performanceaspekt, geeignet.

Classify Benchmark

id	CPU	Ausführungszeit gcc	Ausführungszeit clang	Zeitgewinn durch Clang
Adam	X6 1055T	404	410	-1%
Fabian	i7-4770k	240	239	0%
ISTE_2	i5-3470	309	309	0%
ISTE	i3-2120	818	856	-4%
Patrick	i7-4700MQ	390	390	0%
VGPU1	2x Xeon E5-2620	200	200	0%
SGS	i5-4670	259	259	0%

in Sekunden

Heat Equation Solver

id	CPU	Ausführungszeit gcc	Ausführungszeit clang	Zeitgewinn durch Clang
Adam	X6 1055T	77	86	-10%
Fabian	i7-4770k	58	63	-8%
ISTE_2	i5-3470	66	69	-4%
ISTE	i3-2120	114	114	0%
Patrick	i7-4700MQ	88	71	24%
VGPU1	2x Xeon E5-2620	45	38	18%
SGS	i5-4670	61	61	0%

in Sekunden

Quadratur

id	CPU	Ausführungszeit gcc	Ausführungszeit clang	Zeitgewinn durch Clang
Adam	X6 1055T	72	79	-9%
Fabian	i7-4770k	36	39	-8%
ISTE_2	i5-3470	44	45	-2%
ISTE	i3-2120	65	64	2%
Patrick	i7-4700MQ	43	47	-9%
VGPU1	2x Xeon E5-2620	67	70	-4%
SGS	i5-4670	40	41	-2%

in Sekunden

7 Schlussfolgerungen

Diese Fachstudie hat gezeigt, dass Clang in vielen Anwendungsbereichen ein sehr guter Ersatz für GCC ist. Es wurde gezeigt, dass Clang im Schnitt dreimal so schnell wie GCC kompiliert und die Performanz der Kompilate gleichwertig ist. Die Migration nach Clang hat sich als sehr einfach herausgestellt.

Da die von uns untersuchte Version 3.4 von Clang kein OpenMP unterstützt, ist der Einsatz dieses Compilers trotz aller Vorteile keine Alternative für die *SG*$^{++}$ Bibliothek. Ob Clang 3.5, der OpenMP unterstützen soll, diese Schwachstelle beseitigt, muss im Rahmen nachfolgender Projekte untersucht werden. Die Verwendung von Clang auf Windows ist momentan mangels Funktionalität der Clang Windows Standard Library nur wenig hilfreich.

Beim Vergleich der Toolchains der beiden Compiler war zu erkennen, dass der Clang-Compiler einige Tools liefert die Entwickler unterstützen und ihn produktiver werden lassen. Außerdem wurde gezeigt, dass die Fehlermeldungen besser sind und Fehler leichter erkannt werden können. Die Unterstützung durch frei verfügbare Tools ist für beide Compiler gewährleistet, wie es am Beispiel der Profilingwerkzeuge erarbeitet wurde. Das Ersetzen von GCC durch Clang fällt besonders leicht, da der Clang kompatibel zu den Compilerflags von GCC ist.

So war zum Schluss klar zu erkennen, das Clang durch seinen raschen Aufstieg ein immenses Potential für die Zukunft bietet und man die Entwicklung des Compilers auf jeden Fall nachverfolgen sollte, wenn dieser für die eigenen Zwecke nicht genügt. Ist er geeignet, stellt Clang einen hervorragenden Ersatz für GCC dar.

Literaturverzeichnis

[claa] Clang C++ Features. URL http://clang.llvm.org/cxx_status.html. (Zitiert auf Seite 13)

[clab] Clang Diagnostics Manual. URL http://clang.llvm.org/diagnostics.html. (Zitiert auf Seite 19)

[clac] Clang OpenMP. URL openmp.llvm.org. (Zitiert auf Seite 13)

[clad] Clang OpenMP Implementation. URL https://clang-omp.github.io. (Zitiert auf Seite 13)

[clae] ClangCheck. URL http://clang.llvm.org/docs/ClangCheck.html. (Zitiert auf Seite 20)

[CMH91] P. P. Chang, S. A. Mahlke, W.-m. W. Hwu. Using Profile Information to Assist Classic Code Optimizations. *Softw. Pract. Exper.*, 21(12):1301–1321, 1991. doi: 10.1002/spe.4380211204. URL http://dx.doi.org/10.1002/spe.4380211204. (Zitiert auf Seite 25)

[for] ClangFormat. URL http://clang.llvm.org/docs/ClangFormat.html. (Zitiert auf Seite 21)

[Fou] F. S. Foundation. Internals of the GNU compilers. URL https://gcc.gnu.org/onlinedocs/gccint/. (Zitiert auf Seite 11)

[G+06] J. Garcke, et al. Sparse grid tutorial. *Mathematical Sciences Institute, Australian National University, Canberra Australia*, 2006. (Zitiert auf Seite 7)

[gcca] GCC C++ 11 Features. URL https://gcc.gnu.org/projects/cxx0x.html. (Zitiert auf Seite 13)

[gccb] GCC C++ 14 Features. URL https://gcc.gnu.org/projects/cxx1y.html. (Zitiert auf Seite 13)

[GKM82] S. L. Graham, P. B. Kessler, M. K. Mckusick. Gprof: A Call Graph Execution Profiler. *SIGPLAN Not.*, 17(6):120–126, 1982. doi:10.1145/872726.806987. URL http://doi.acm.org/10.1145/872726.806987. (Zitiert auf Seite 29)

[gnu] GNU Compiler Collection. URL https://gcc.gnu.org/. (Zitiert auf Seite 7)

[gooa] Google Performance Tools. URL https://code.google.com/p/gperftools/wiki/GooglePerformanceTools. (Zitiert auf Seite 29)

[goob] Google Performance Tools CPU Profile. URL http://google-perftools.googlecode.com/svn/trunk/doc/cpuprofile.html. (Zitiert auf Seite 29)

[gpra] Gprof Manual Pages. URL http://linux.die.net/man/1/gprof. (Zitiert auf Seite 29)

[gprb] Gprof Utah. URL https://www.cs.utah.edu/dept/old/texinfo/as/gprof.html. (Zitiert auf Seite 29)

[icc] Intel C and C++ Compilers. URL https://software.intel.com/en-us/c-compilers. (Zitiert auf Seite 7)

[ker] Linux Kernel Intel Perf Events. URL https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/arch/x86/kernel/cpu/perf_event_intel.c?id=dea4f48a0a301b23c65af8e4fe8ccf360c272fbf#n2367. (Zitiert auf Seite 28)

[Lat08] C. Lattner. LLVM and Clang: Next generation compiler technology. In *The BSD Conference*, S. 1–2. 2008. (Zitiert auf Seite 11)

[mem] MemorySanitizer. URL http://clang.llvm.org/docs/MemorySanitizer.html. (Zitiert auf Seite 22)

[moda] Clang Modernize Usage. URL http://clang.llvm.org/extra/ModernizerUsage.html. (Zitiert auf Seite 21)

[modb] Clang Modernize User Manual. URL http://clang.llvm.org/extra/clang-modernize.html. (Zitiert auf Seite 21)

[per] Sampling with Perf. URL https://perf.wiki.kernel.org/index.php/Tutorial#Sampling_with_perf_record. (Zitiert auf Seite 27)

[PPB10] D. Pflüger, B. Peherstorfer, H.-J. Bungartz. Spatially adaptive sparse grids for high-dimensional data-driven problems. 2010. (Zitiert auf den Seiten 7, 9 und 10)

[SBPV12] K. Serebryany, D. Bruening, A. Potapenko, D. Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX Annual Technical Conference*, S. 309–318. 2012. (Zitiert auf Seite 23)

[SI09] K. Serebryany, T. Iskhodzhanov. ThreadSanitizer: data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, S. 62–71. ACM, 2009. (Zitiert auf Seite 23)

[sta] Clang Static Analyzer. URL `http://clang-analyzer.llvm.org/`. (Zitiert auf Seite 22)

[too] Clang Lib Tooling. URL `http://clang.llvm.org/docs/LibTooling.html`. (Zitiert auf Seite 20)

[vala] Valgrind Anleitung. URL `http://valgrind.org/docs/manual/cg-manual.html`. (Zitiert auf Seite 29)

[valb] Valgrind Core Manual. URL `http://valgrind.org/docs/manual/manual-core.html`. (Zitiert auf Seite 28)

Alle URLs wurden zuletzt am 20. Oktober 2014überprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift