**시스템 프로그램**

**Project-1**

**컴퓨터공학과 2016310487 최세현**

**컴퓨터공학과 2016312160 이준성**

**선정 프로그램**

시스템 프로그래밍 실습 수업에서 개발했던 문자열 처리 프로그램

**선정 프로그램 설명**

1. indexBuilder.c

임의의 기존 binary데이터파일(dataStorage)을 읽어들여 기존 정보를 불러들인 후

프로그램 실행 시 입력된 파라메터(텍스트파일 이름)로, 여러 종류의 ‘성경’ 텍스트파일들을 읽어들여

모든단어들의 개수와 위치를 각각 성경별로 정보 저장, 임의의 binary데이터파일에 정보 출력

1. indexPrinter.c

임의의 기존 binary데이터파일(dataStorage)을 읽어들여 정보를 분석 후

프로그램 실행 시 입력된 파라메터(텍스트파일 이름)로 텍스트파일 생성 후 텍스트형태로 모든 정보 출력

1. main.c

위 두 소스코드를 이용해 입력된 모든 성경파일의 단어를 분석해서 텍스트형태로 텍스트파일에 출력시킴

텍스트이(가) 표시된 사진

자동 생성된 설명

**선정 프로그램 실행방법**

-Og -pg 옵션으로 3개의 소스파일 컴파일

./indexBP (파악 할 성경 텍스트파일들 이름) (출력할 텍스트파일 이름) 로 실행텍스트이(가) 표시된 사진

자동 생성된 설명

위와 같이 실행 시genesis.txt부터 malachi.txt까지 41개의 성경텍스트파일의 분석결과가 result.txt에 저장,

총 실행시간이 usec단위로 나타남

성경 파일은 <http://www.stewartonbibleschool.org.uk/bible/text/> 에서 다운로드

Optimization과 분석은 위 사진과 똑같이 실행하여 진행

여러 번 실행하여 가장 평균적인 실행시간과 분석결과가 나오는 것으로 보고서 작성

**First Profiling**

**프로그램 실행시간 : 33초(33948867(usec))**

텍스트이(가) 표시된 사진

자동 생성된 설명

**1차 gprof 분석결과 :**

기본 C라이브러리 함수들을 많이 이용하는 프로그램이다 보니, 실제 함수들의 실행시간은 총1.47초로 작다. 또한 대부분의 함수들 각각의 실행기간은 작지만 몇몇 함수들의 호출량이 매우 많아 실행시간이 늘어난 것으로 보인다.

**1차 개선점 분석 :**

호출량이 많은 함수들 사이에서 loop내에서 사용되는 많은 연산을 필요로 하지 않는getBibleNumber, getCurrentBilbleNumber 같은 함수들이 꽤 많은 시간을 차지하고 있는 것이 보인다. 이 함수들은 loop 조건문에서 필요이상으로 많이 호출되므로, 제거되어야 한다.

**Optimization 1 – loop optimization**

**변경 후 실행시간 : 33초 🡪 31초 (31834783(usec))**

**대상 :** getBibleNumber(), getCurrentBibleNumber()함수

**원인 :** for loop의 조건문에서 불필요한 반복 호출

**개선 :** 함수를 삭제하고, 직접 접근하도록 코드를 모두 변경했다.

**before :** **for**(**int** i = 0; i < getBibleNumber(); i++)

**after :** **for**(**int** i = 0; i < bibleNumber; i++)

**before :** **for**(**int** i = 0; i < getCurrentBibleNumber(); i++)

**after :** **for**(**int** i = 0; i < bibleNumber - 1; i++)

**Second Profiling**

텍스트이(가) 표시된 사진

자동 생성된 설명

**2차 gprof 분석**

특히 FilePrintAllTrie 함수와, freeAllTrie 함수에서 많은 사용을 했었는데, 함수자체의 삭제이기 때문에, gprof 분석결과에서 변경점을 찾기 힘들지만 총 실행시간 1.47초에서 1.35초로의 유의미한 감소를 확인할 수 있다. 하지만 여전히 호출량이 많은 다른 함수들이 시간을 적지않게 차지하고 있다.

**2차 개선점 분석**

대부분 코딩의 편의와 가독성을 위해 사용한 get/set함수이므로, 함수를 제거하고 직접 접근한다면 반복적인 호출을 없애 시간을 줄일 수 있을 것이다.

**Optimization 2 – reducing procedure calls**

**변경 후 실행시간 : 31초 🡪 24초 (24555098(usec))**

**대상 :** 가독성과 코딩의 편의를 위해 사용했던 많은 get/set 함수

**원인 :** 불필요한 대량의 함수호출로 인한 엄청난 실행시간의 증가가 유발되므로 삭제되어야 한다.

**개선 :** 모든 편의를 위한 get/set 함수를 삭제하고, 함수호출부분을 직접 접근하는 방식의 코드로 변경한다.

**EX) TrieInsertData함수의 일부분 (변경함수 : main, TrieInsertData, freeAllTrie, fileWriteAllTrie)**

**before :**

setEndOfWord(t);

**if**(getData(t) == **NULL**)

{

setData(t, malloc(**sizeof**(word) \* getBibleNumber(&d)));

**for**(**int** i = 0; i < getCurrentBibleNumber(&d); i++)

setDataWordNumber(t, i, 0);

setCurrentDataWordNumber(&d, t, 0);

setCurrentDataIndex1(&d, t, malloc(**sizeof**(**int**) \* 100));

setCurrentDataIndex2(&d, t, malloc(**sizeof**(**int**) \* 100));

setCurrentDataPos(&d, t, malloc(**sizeof**(**int**) \* 100));

setCurrentDataMemAlloc(&d, t, 1);

}

{ . . . }

**if**(getCurrentDataWordNumber(&d, t) >= 100 \* getCurrentDataMemAlloc(&d, t))

{

setCurrentDataMemAlloc(&d, t, getCurrentDataMemAlloc(&d, t)+1);

setCurrentDataIndex1(&d, t, realloc(getCurrentDataIndex1(&d, t), **sizeof**(**int**) \* (100 \* getCurrentDataMemAlloc(&d, t))));

setCurrentDataIndex2(&d, t, realloc(getCurrentDataIndex2(&d, t), **sizeof**(**int**) \* (100 \* getCurrentDataMemAlloc(&d, t))));

setCurrentDataPos(&d, t, realloc(getCurrentDataPos(&d, t), **sizeof**(**int**) \* (100 \* getCurrentDataMemAlloc(&d, t))));

}

insertDataToCurrent(&d, t, index1, index2, pos);

**after :**

t->is\_terminal = **true**;

**if**(t->data == **NULL**)

{

t->data = malloc(**sizeof**(word) \* bibleNumber);

**for**(**int** i = 0; i < bibleNumber-1; i++)

t->data[i].wordNumber = 0;

t->data[bibleNumber-1].wordNumber = 0;

t->data[bibleNumber-1].index1 = malloc(**sizeof**(**int**) \* 100);

t->data[bibleNumber-1].index2 = malloc(**sizeof**(**int**) \* 100);

t->data[bibleNumber-1].pos = malloc(**sizeof**(**int**) \* 100);

t->data[bibleNumber-1].memAlloc = 1;

}

{ . . . }

**if**(t->data[bibleNumber-1].wordNumber >= 100 \* t->data[bibleNumber-1].memAlloc)

{

t->data[bibleNumber-1].memAlloc++;

t->data[bibleNumber-1].index1 = realloc(t->data[bibleNumber-1].index1, **sizeof**(**int**) \* (100 \* t->data[bibleNumber-1].memAlloc));

t->data[bibleNumber-1].index2 = realloc(t->data[bibleNumber-1].index2, **sizeof**(**int**) \* (100 \* t->data[bibleNumber-1].memAlloc));

t->data[bibleNumber-1].pos = realloc(t->data[bibleNumber-1].pos, **sizeof**(**int**) \* (100 \* t->data[bibleNumber-1].memAlloc));

}

t->data[bibleNumber-1].index1[t->data[bibleNumber-1].wordNumber] = index1;

t->data[bibleNumber-1].index2[t->data[bibleNumber-1].wordNumber] = index2;

t->data[bibleNumber-1].pos[t->data[bibleNumber-1].wordNumber] = pos;

t->data[bibleNumber-1].wordNumber++;

**Third Profiling**

전자기기, 계산기이(가) 표시된 사진

자동 생성된 설명

**3차 gprof 분석**

모든 불필요한 get/set함수들이 사라진 것을 확인할 수 있고, 전체적인 실행시간 감소(1.35 🡪 1.24초) 또한 확인할 수 있다. 이제 주요함수들만 남았다. 함수 내에 불필요한 반복호출이나 메모리 참조는 없는지 확인해보자.

**3차 개선점 분석**

상위 주요함수들 가운데 indexChar 함수에 눈에 띈다. 많은 양의 호출과 아래의 더 중요도 높은 함수보다도 많은 시간을 차지 하고 있다. indexChar 함수는 메모리에 직접 접근해 값을 바꾼다. 메모리 참조횟수를 줄인다면 시간을 더욱 줄일 수 있을 것이다.

**Optimization 3 – reducing memory references**

**변경 후 실행시간 : 24초 🡪 23초 (23673391(usec))**

**대상 :** TrieInsertData/Empty 함수에서 indexChar 함수의 이용

**원인 :** 읽어온 문자를 Trie 자료구조에 저장을 위한 인덱스로 변경하는 indexChar함수는,

메모리 참조를 통해 문자의 값 자체를 바꿔주는 기능을 한다.

TrieInsertData/Empty 함수에서 indexChar함수로 해당 메모리값을 변경한 후 다시 반복해서 메모리 값을

직접 읽는 연산을 수행하기 때문에 많은 양의 불필요한 메모리 참조가 일어나게 된다.

**개선 :** 실행시간의 감소를 위해 메모리 참조를 하지 않고 지역변수를 이용해

레지스터에서 직접 참조하도록 변경한다.

**before :**

**void** indexChar(**char** \*c)

{

**if**(\*c >= 'A' && \*c <= 'Z')

\*c -= 63;

**else** **if**(\*c >= 'a' && \*c <= 'z')

\*c -= 95;

**else** **if**(\*c == '-')

{ . . . }

}

indexChar(key);

**if**(t->children[\*key] == 0)

t->children[\*key] = createTrie();

TrieInsertData(t->children[\*key], key + 1, index1, index2, pos);

**after :**

**char** indexChar(**char** c)

{

**if**(c >= 'A' && c <= 'Z')

**return** c - 63;

**else** **if**(c >= 'a' && c <= 'z')

**return** c - 95;

**else** **if**(c == '-')

{ . . . }

}

**int** index = indexChar(\*key);

**if**(t->children[index] == 0)

t->children[index] = createTrie();

TrieInsertData(t->children[index], key + 1, index1, index2, pos);

**Fourth Profiling**

전자기기, 계산기이(가) 표시된 사진

자동 생성된 설명

**4차 gprof 분석**

indexChar함수 실행시간의 감소(0.04 🡪 0.01)와 전체적인 시간감소(1.24 🡪 1.05)를 확인할 수 있다. 다른 주요함수들 중에서도 불필요한 반복이나 참조는 없는지 확인해보자.

**4차 개선점 분석**

상위 함수들 중 28번의 반복을 수행하는 for loop을 포함하는 freeAllTrie 함수가 많은 호출과 시간을 차지하고 있다. for loop 의 unrolling작업을 통해 loop 반복 횟수를 줄여 개선할 수 있을 것이다.

**Optimization 4 – loop unrolling**

**변경 후 실행시간 : 23초 🡪 22초 (22911477(usec))**

**대상 :** freeAllTrie함수의 for loop

**원인 :** Trie 자료구조의 메모리 해제를 위해 for loop을 28번 돌려 28번의 loop이 실행된다

**개선 :** 7\*4로, 7way loop unrolling을 통해 loop횟수를 28번에서 4번으로 줄여 비교연산, jump횟수 등을 줄인다.

이 작업을 통해 indexBuilder/Printer 모두에서 실시하는 메모리 해제작업을 담당하고 있고,

또, 실행시간의 큰 비중을 차지하고 있는 freeAllTrie함수의 실행시간을 감소시킬 수 있다.

**before :**

**for**(**int** i = 0; i < 28; i++)

{

**if**(t->children[i])

freeAllTrie(t->children[i], bibleNumber);

}

**after :**

**for**(**int** i = 0; i < 28; i+=7)

{

**if**(t->children[i])

freeAllTrie(t->children[i], bibleNumber);

**if**(t->children[i+1])

freeAllTrie(t->children[i+1], bibleNumber);

**if**(t->children[i+2])

freeAllTrie(t->children[i+2], bibleNumber);

**if**(t->children[i+3])

freeAllTrie(t->children[i+3], bibleNumber);

**if**(t->children[i+4])

freeAllTrie(t->children[i+4], bibleNumber);

**if**(t->children[i+5])

freeAllTrie(t->children[i+5], bibleNumber);

**if**(t->children[i+6])

freeAllTrie(t->children[i+6], bibleNumber);

}

**Fifth Profiling**

전자기기이(가) 표시된 사진

자동 생성된 설명

**5차 gprof 분석**

freeAllTrie의 실행시간 감소(0.21 🡪 0.18)와 전체적인 실행시간감소(1.05 🡪 0.96)를 확인할 수 있다.

**5차 개선점 분석**

FilePrintAllTrie 함수를 제외한 모든 함수에서의 Optimization과정을 마쳤다.

FilePrintAllTrie 함수는 저장한 모든 정보를 참조하여 파일에 출력하는 기능을 하기 때문에 시간이 많이 걸릴 수

밖에 없고, Optimize 하기에 한계가 있다.

이로써 할 수 있는 모든 Optimize 과정을 실시했고, 더 이상의 Optimization은 중단한다.

**<결론>**

**Optimization 1 – loop optimization**

변경 후 실행시간 : 33초 🡪 31초 (31834783(usec))

gprof 전체 실행시간 : 1.47초 🡪 1.35초

요약 : 함수자체의 삭제이기 때문에 gprof에서 변경점을 찾기 힘들지만 전체 실행시간의 감소 확인

**Optimization 2 – reducing procedure calls**

변경 후 실행시간 : 31초 🡪 24초 (24555098(usec))

gprof 전체 실행시간 : 1.35초 🡪 1.24초

요약 : 가장 많은 실행시간 감소가 일어난 단계, 엄청난 양의 함수호출을 감소시킴

**Optimization 3 – reducing memory references**

변경 후 실행시간 : 24초 🡪 23초 (23673391(usec))

gprof 전체 실행시간 : 1.24초 🡪 1.05초

요약 : memory참조를 줄여 한 함수의 유의미한 실행시간 감소를 확인할 수 있음

**Optimization 4 – loop unrolling**

변경 후 실행시간 : 23초 🡪 22초(22911477(usec))

gprof 전체 실행시간 : 1.05초 🡪 0.96초

요약 : 28번의 loop를 7\*4로 unrolling

**결과**

프로그램 실행 시간 : 33초 🡪 22초

gprof기준 전체 함수 실행시간 : 1.47초 🡪 0.96초

**한계**

1. 문자열의 처리를 수행하는 프로그램으로서 계산, 연산을 주로 수행하기 보다는, 문자열 처리와 자료구조로의 저장이 주로 이루어졌기 때문에, 병렬적인 연산수행, Parallelism optimization를 적용하기에 한계가 있었다.
2. 단어의 길이가 분명하지 않기에 단어의 저장에 사용되는 Trie자료구조 연산을 for loop으로 구현하여 optimize하기에는 한계가 있어서 많은 양의 재귀호출, procedure call을 사용할 수 밖에 없었다.
3. 파일입출력과 문자열 처리를 주로 수행하는 프로그램이라, 기본 C 라이브러리 함수가 많은 시간을 차지하기 때문에 optimize 후의 큰 함수 실행시간 감소를 기대하긴 힘들다.