Project-1 Report

SYSTEM PROGRAMMING

Program Profiling & Optimizing

과목	시스템프로그램
담당교수	엄영익
제출일	2019.12.11
성명 / 학번	최욱철 / 2015312560
학과	컴퓨터공학과
	김민호 / 2013313671
	시스템경영공학과

1. 선정한 프로그램의 개요

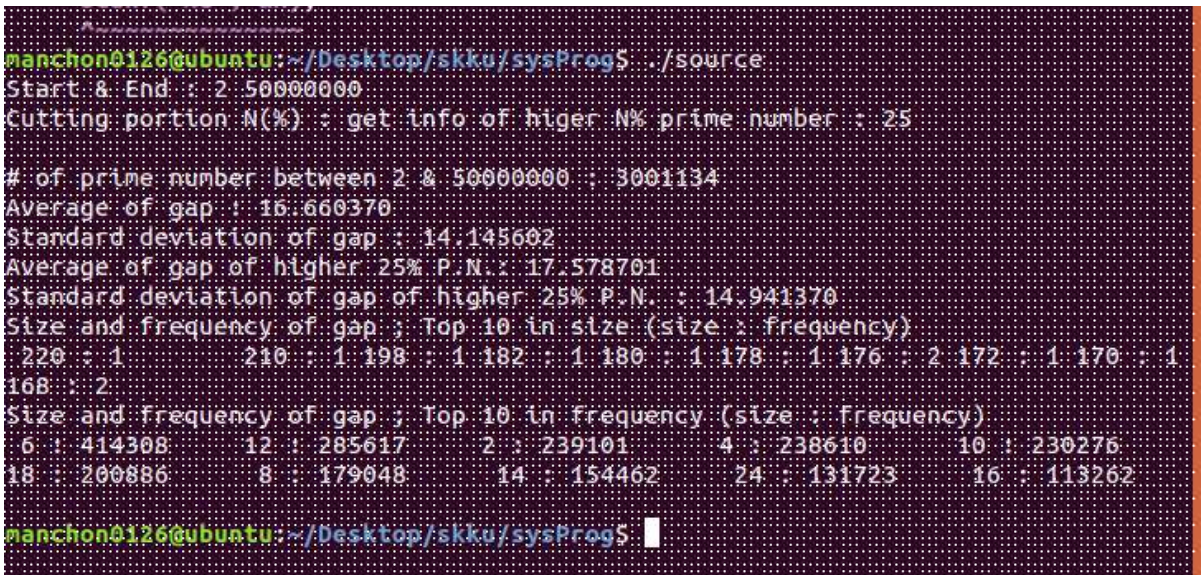
본 프로그램은 소수(prime number)의 분포의 성질에 대해 분석하기 위한 프로그램입니다. 사용자는 2 이상의 두 정수 X, Y를 입력한 후 다음 라인에 0이상 100이하의 정수 N을 입력합니다. 프로그램은 ① X이상 Y이하의 소수의 개수를 출력하고(# of prime number between X & Y), ② "그 소수들 사이의 간격"의 평균을 출력하고(Average of gap)(예를 들어 X=2, Y=8일 경우 범위내의 소수는 2,3,5,7이고 소수 사이의 간격은 1,2,2 이므로 1.666667을 출력하는 것입니다), ③ "소수들 사이의 간격"의 표준편차를 출력하고(Standard deviation of gap), ④⑤ X이상 Y이하 범위 중 크기가 큰 N퍼센트의 소수에 대한 평균과 표준편차를 출력하고 (~of higher N% P.N.), ⑥⑦ "소수들 사이의 간격"은 같은 값이 대단히 많이 나올 것이므로 그 중에서 크기가 가장 큰 10개는 무엇인지 "간격의 크기 : 출현 횟수" 순으로 출력하고, 빈도가 가장 큰 10개는 무엇인지 ⑥과 동일한 format으로 출력합니다. (④⑤와 ⑥의 차이는, ④⑤는 "원래 소수"의 크기를 가지고 상위 N%만을 택하는 것이고 ⑥은 "소수들 사이의 간격"의 크기를 가지고 상위 10개를 택하는 것입니다)

2. 프로그램 컴파일 및 실행 방법

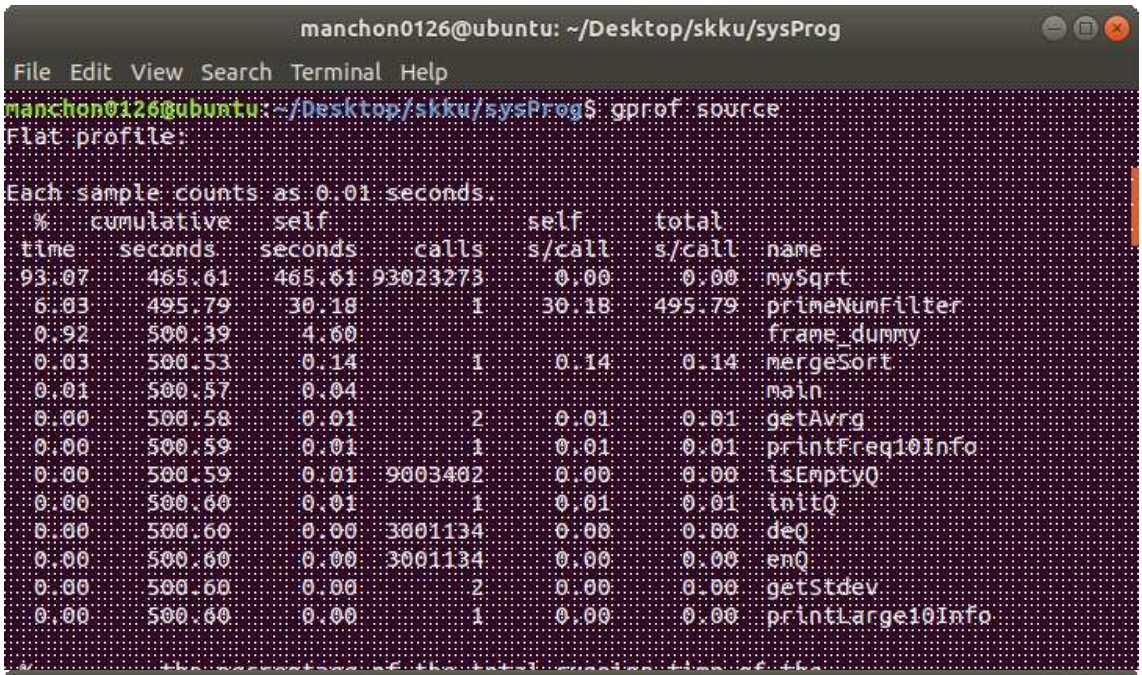
모든 optimization 단계에 대해 강의 교재의 5.14.1단원에서 제시된, gcc -Og -pg xxxx.c -o xxxx 옵션을 적용했고 입력 값은 X=2, Y=50,000,000, N=25로 일괄 적용했습니다.

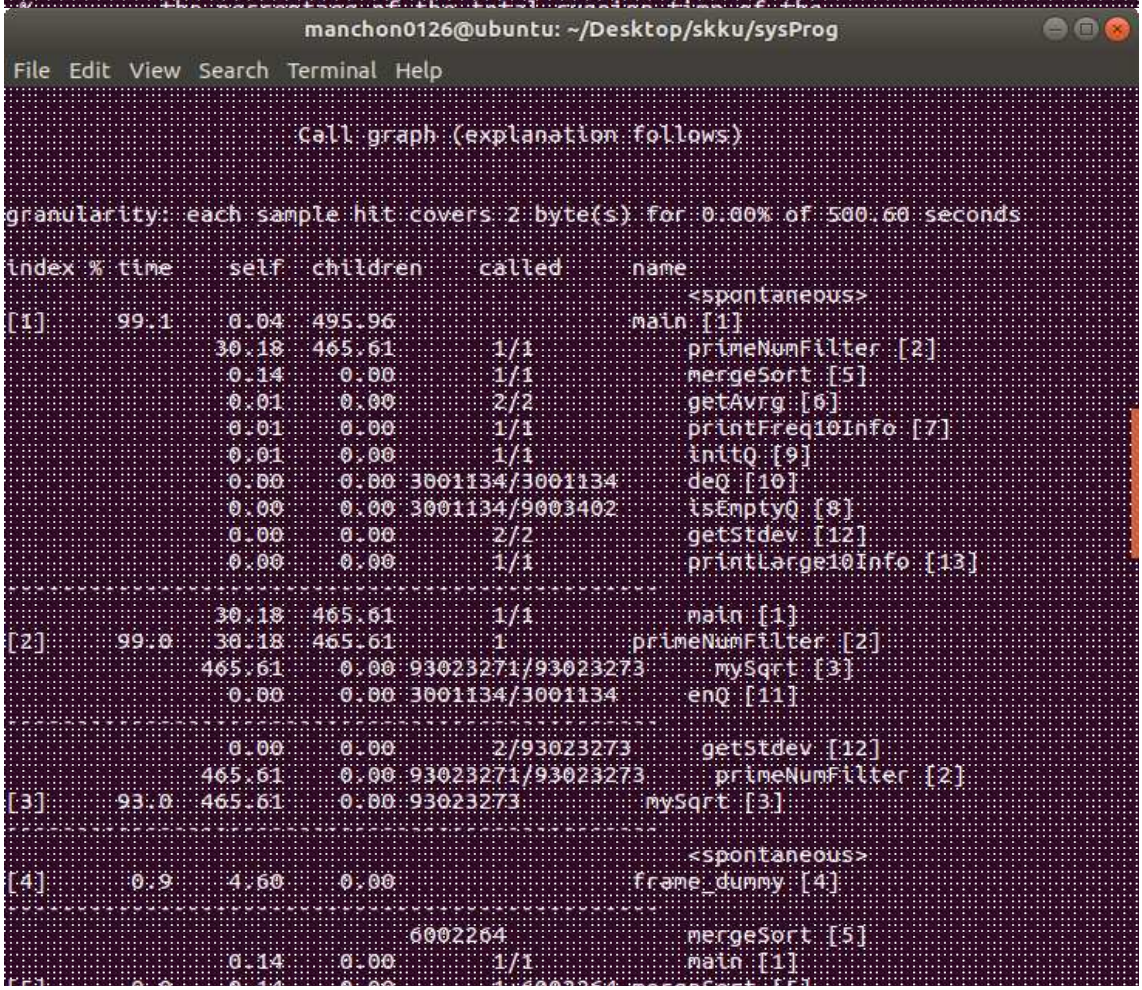
3. Optimization 과정

우선 어떠한 수정도 하지 않은 상태인 source.c 파일을 컴파일 후 실행했습니다.
실행 결과는 아래와 같습니다.



GPROF로 분석한 결과는 다음과 같습니다.





전체 실행시간 500.60초 중 함수 mySqrt가 465.61초로 90%이상을 차지하고 있음을, 그리고 단 2회를 제외하고는 "primeNumFilter"가 호출했음도 알 수 있습니다. 이를 보면 최우선적으로 optimize해야 할 함수로 지정해야 함이 분명합니다. primeNumFilter는 A이상 B이하의 정수 중 소수만을 걸러내는 함수로, for문을 통해 일일이 나눠봄으로서 2와 자신을 제외한 약수가 있는지 확인하는 방법을 사용하고 있습니다. 이때 계산 수를 줄이기 위해 "2이상의 자연수 n에 대해 $\sqrt{n}$이하의 약수가 존재하지 않는다면 n은 소수"라는 점을 이용해 $\sqrt{n}$이상의 수로는 나눠보지 않도록 한 것인데, 이것이 문제점이 된 것입니다. source.c는 for문을 돌 때 마다 mySqrt함수를 호출하게 되는 비효율적인 구조를 가지고 있습니다. 따라서 mySqrt 함수를 for문 밖으로 빼냈습니다. source_2.c 입니다. 변경된 부분은 다음과 같습니다.

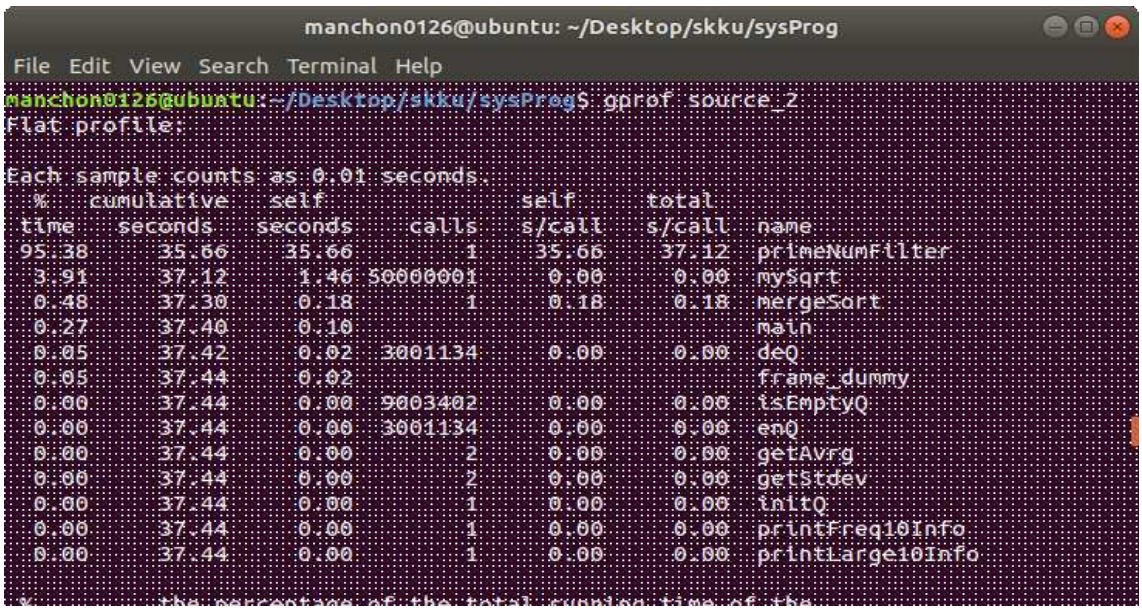
변경 전(source)

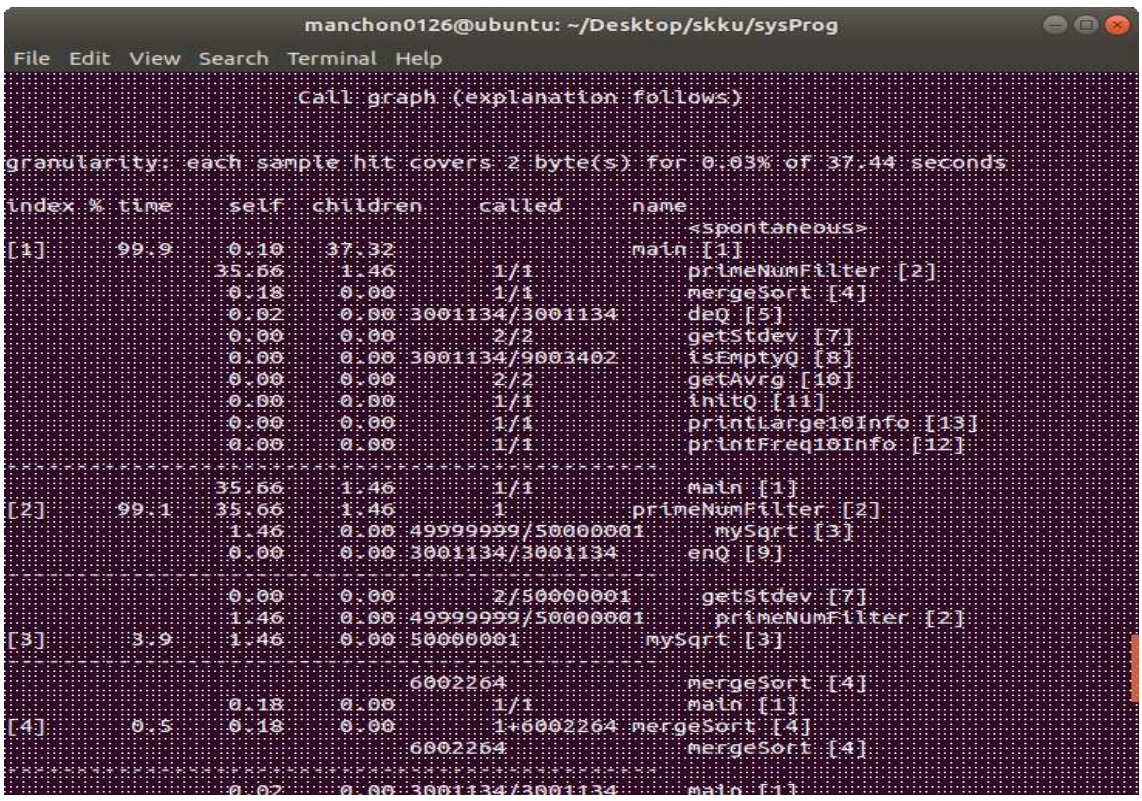
```
int primeNumFilter(Queue* pq, int A, int B){
    int countPN = 0;
    int previousPN = 0;

    for(int i = A; i<=B; i++){
        bool isPrime = true;
        for(int j = 2; j<= mySqrt(i); j++){
            if(i%j == 0){
                isPrime = false;
                break;
            }
        }
        if(isPrime){
            enQ(pq, i-previousPN);
            previousPN = i;
            countPN++;
        }
    }

    return countPN;
}
```

변경 후(source_2)

```
int primeNumFilter(Queue* pq, int A, int B){
    int countPN = 0;
    int previousPN = 0;

    for(int i = A; i<=B; i++){
        bool isPrime = true;
        int sqrtI = (int) mySqrt(i);
        for(int j = 2; j<= sqrtI; j++){
```

```
            if(i%j == 0){
                isPrime = false;
                break;
            }
        }

        (생략)
```





source_2로 optimize한 결과 전체 mySqrt의 총 실행시간은 465.61에서 1.46초로 무려 319배가량 빨라져 실행시간은 500.60초에서 37.44초로 13배가량 빨라졌습니다. 이제 primeNumFilter가 전체 실행시간의 95%가량을 차지하게 됐으므로, 이 함수를 optimize할 타깃으로 삼았습니다. source_2에서 안쪽의 for문은 약수가 있는지 체크하는 과정으로, 약수가 발견되면 isPrime을 false로 변환시키고 break합니다. 만약 약수가 없이 for문을 완전히 돌면 아래의 if(isPrime)은 참이므로 queue에 바로 이전에 발견된 소수와의 차를 enqueue하는 것입니다. 이때 아랫쪽 if문을 없앨 경우 불필요한 분기를 하나 제외시킬 수 있고 따라서 좀 더 효율적으로 동작할 것을 기대할 수 있을 것으로 보입니다. 그래서 일반적으론 권장되지는 않는 goto문을 사용해 아래와 수정했습니다. source_3입니다.

변경 전(source_2)

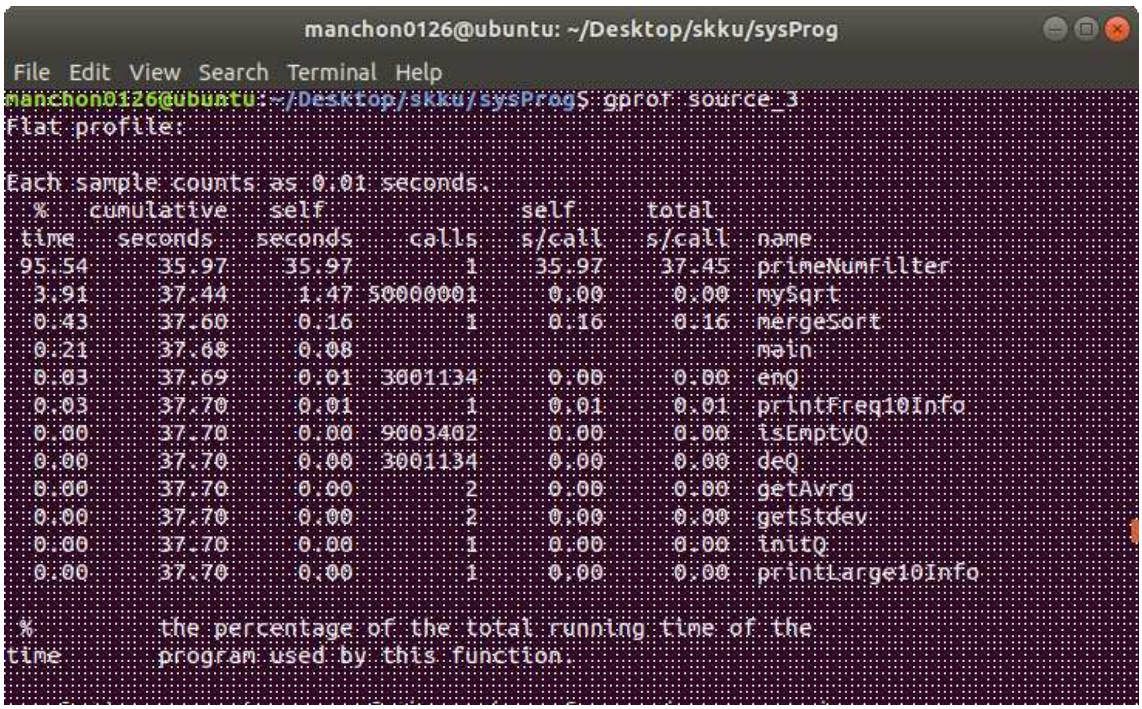
```
for(int i = A; i<=B; i++){
    bool isPrime = true;
    int sqrtI = (int) mySqrt(i);
    for(int j = 2; j<= sqrtI; j++){
        if(i%j == 0){
            isPrime = false;
            break;
        }
    }
    if(isPrime){
        enQ(pq, i-previousPN);
        previousPN = i;
        countPN++;
    }
}
```

변경 후(source_3)

```
int i = A;

primeLoop:
if(i<=B){
    int sqrtI = (int) mySqrt(i);
    for(int j = 2; j<= sqrtI; j++){
        if(i%j == 0){
            i++;
            goto primeLoop;
        }
    }
    enQ(pq, i-previousPN);
    previousPN = i;
    countPN++;
```

```
    i++;
    goto primeLoop;
}
```

실행 결과는 아래와 같습니다.



primeNumFilter의 실행시간은 35.66초에서 35.97초로 오히려 1.01배가량 늘었으며 이에 따라 전체 실행시간도 37.44에서 37.70초로 늘어났습니다. 이미 processor들의 prefetching과 branch prediction이 성공적으로 시행되고 있었으리라 추측 가능합니다. 이번에는 loop unrolling을 시도해 봤습니다. source_4입니다.

변경 전(source_3)

```
int primeNumFilter(Queue* pq, int A, int B){
    int countPN = 0;
    int previousPN = 0;
    int i = A;

    primeLoop:
    if(i<=B){
        int sqrtI = (int) mySqrt(i);
        for(int j = 2; j<= sqrtI; j++){
            if(i%j == 0){
```

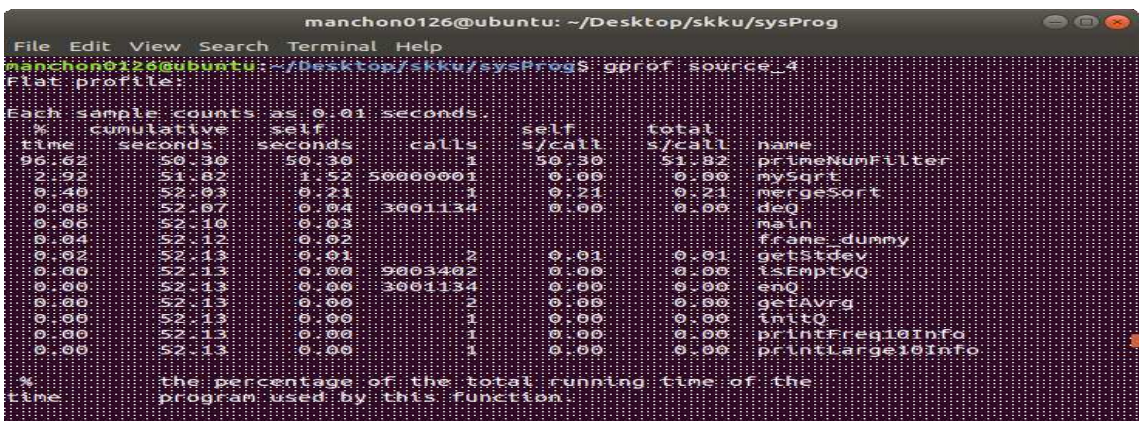
```
                i++;
                goto primeLoop;
            }
        }
        enQ(pq, i-previousPN);
        previousPN = i;
        countPN++;
        i++;
        goto primeLoop;
    }

    return countPN;
}
```

변경 후(source_4)

```
int primeNumFilter(Queue* pq, int A, int B){
    int countPN = 0;
    int previousPN = 0;

    int i = A;
    int limit = B-1;

    for( ; i<=limit; i+=2){
        bool isPrime0 = true, isPrime1 = true;
        int sqrtI0 = (int) mySqrt(i), sqrtI1 = (int) mySqrt(i+1);
        int i1 = i+1;
        int j =2;

        while( (isPrime0 || isPrime1 ) && j<= sqrtI1){
            if( isPrime0 ) {
                isPrime0 = (i%j);
            }
            if( isPrime1 ){
                isPrime1 = (i1%j);
            }
            j++;
        }



        if(isPrime0){
            enQ(pq, i-previousPN);
            previousPN = i;
```

```
            countPN++;
        }
        if(isPrime1){
            enQ(pq, i1-previousPN);
            previousPN = i1;
            countPN++;
        }
    }

    for( ; i<=B; i++){
        bool isPrime = true;
        int sqrtI = (int) mySqrt(i);
        for(int j = 2; j<= sqrtI; j++){
            if(i%j == 0){
                isPrime = false;
                break;
            }
        }
        if(isPrime){
            enQ(pq, i-previousPN);
            previousPN = i;
            countPN++;
        }
    }

    return countPN;
}
```

안쪽의 while에서 매 번의 loop를 돌 때 i,i1 두 개의 변수에 대해 %연산을 하고, 소수인지 여부를 저장하는 변수도 isPrime0와 isPrime1로 나눠서 superscalar를 통한 parallelity의 향상이 있을지 시도해보는 것입니다. 실행 결과는 아래와 같습니다.



primeNumFilter의 실행시간이 source_2에서의 35.66초에서 50.30초로 1.41배 정도 늘어났습니다. 본 프로그램을 실행한 machine의 나누기 연산이 파이프라인화 되지 않았거나 제대로 사용되지 못하고 있는 것으로 보입니다. 이 방법은 취소하는 게 확실히 좋겠지만, enQ과정에서 i-previousPN, i1-previousPN와 같이 이전 소수를 계산 후 그 값을 빼야 enqueue를 할 수 있다는 점이 bottleneck으로 작용한 것이 아닐까 하는 의심이 들어 그 부분도 함께 수정했습니다. 소수 사이의 차를 enqueue시키는 방법이 아니라, 소수를 그대로 enqueue시킨 후 main함수에서 dequeue시키는 과정에 소수 사이의 차를 계산하는 방식으로 변경하면 어떻게 될 까 하는 것입니다. 다음은 source_5입니다.

변경 전(source_4)

```
(생략)
if(isPrime0){
        enQ(pq, i-previousPN);
        previousPN = i;
        countPN++;
    }
    if(isPrime1){
        enQ(pq, i1-previousPN);
        previousPN = i1;
         countPN++;
    }
}

for( ; i<=B; i++){
    bool isPrime = true;
    int sqrtI = (int) mySqrt(i);
     for(int j = 2; j<= sqrtI; j++){
        if(i%j == 0){
            isPrime = false;
            break;
        }
    }
    if(isPrime){
       enQ(pq, i-previousPN);
       previousPN = i;
       countPN++;
    }
}
```

변경 후(source_5)

```
(생략)
if(isPrime0){
```

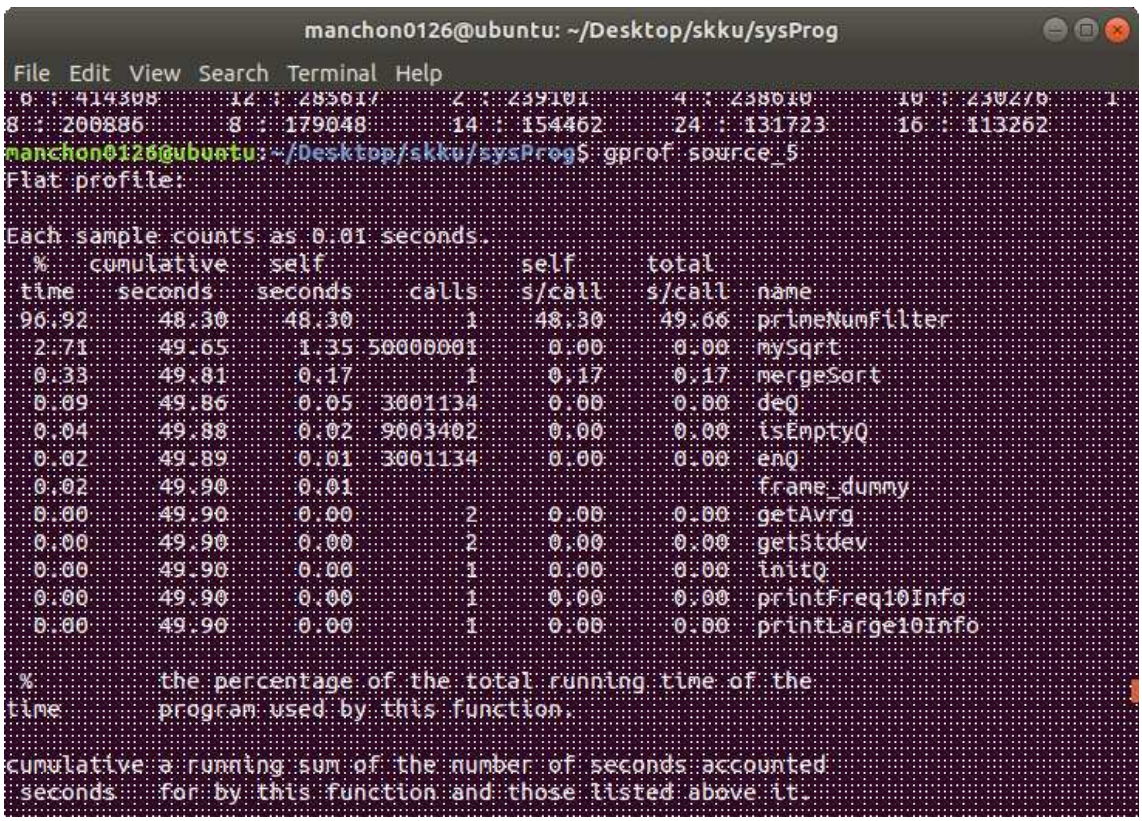
```
            enQ(pq, i);
            countPN++;
        }
        if(isPrime1){
            enQ(pq, i1);
            countPN++;
        }
}

for( ; i<=B; i++){
        bool isPrime = true;
        int sqrtI = (int) mySqrt(i);
        for(int j = 2; j<= sqrtI; j++){
            if(i%j == 0){
                isPrime = false;
                break;
            }
        }
        if(isPrime){
            enQ(pq, i);
            countPN++;
        }
}
```

enqueue 과정은 상기와 같이 바꾸고, 이에 따라 main 함수 내에서 dequeue하는 과정도 변경했습니다.

```
변경 전(source_4)
if( numOfPN >= 1){
    deQ(&myQ);
}

(중략)

while( !isEmptyQ(&myQ) ){
    Data temp = deQ(&myQ);
    arrayOfGap[idx] = temp;
    idx++;
}
```

```
변경 후(source_5)
if( numOfPN >= 1){
    previousPN = deQ(&myQ);
}

(중략)

while( !isEmptyQ(&myQ) ){
    Data temp = deQ(&myQ);
    arrayOfGap[idx] = temp - previousPN;
    previousPN = temp;
    idx++;
}
```

실행 결과는 아래와 같습니다.

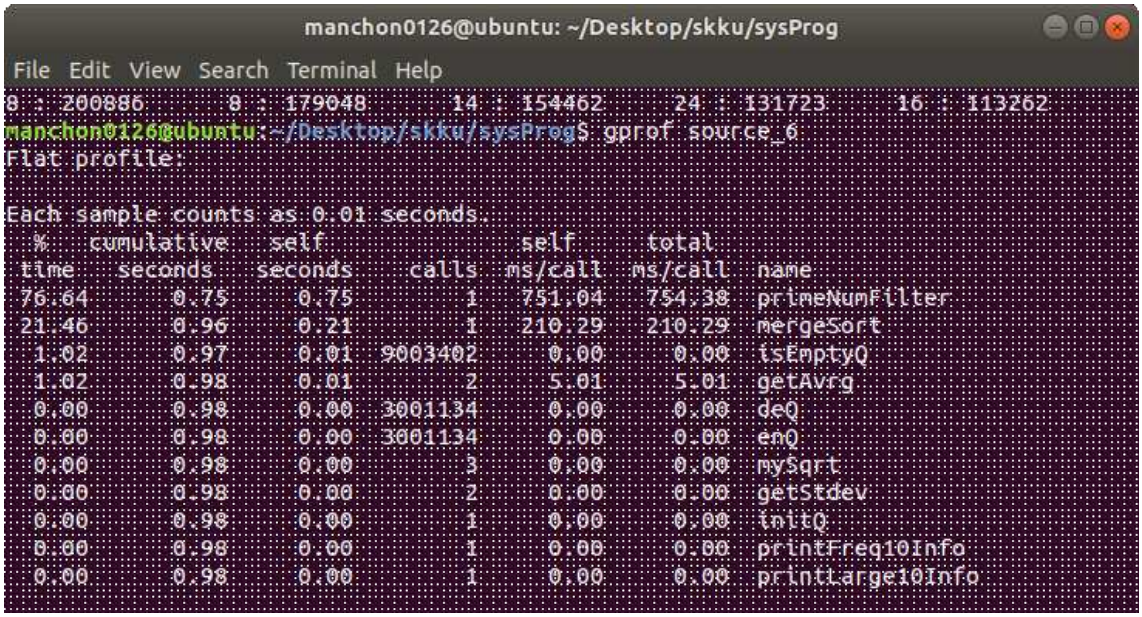


primeNumFilter의 실행시간은 50.30초에서 48.30초로 2초 줄어 1.04배 빨라졌고, 전체 실행시간은 52.13초에서 49.90초로 2.23초 줄었습니다. enqueue과정을 단순화한 대신 dequeue과정이 복잡해졌으므로 전체 실행시간이 primeNumFilter에서 단축된 시간보다 더 많이 단축되었다는 점은 GPROF의 분석 상에 오차가 일부 있었다고 해석하는 편이 좀 더 타당하겠습니다만, 미소하지만 개선 사항이 있었다고 보는 편이 맞겠습니다. 따라서 source_3, 4에서의 수정사

항은 폐기하고, source_5의 수정사항은 취하도록 하겠습니다. 또한 생각해보자면, source_3, 4에서와 같이 불필요한 것으로 보이는 분기의 삭제, unrolling 시도 등이 사실상 효과가 없거나 오히려 역효과를 보이며 source_5와 같이 상대적으로 준수하다고 생각되는 수정사항 역시 단 1.04배 밖에 효과를 보지 못했다는 점에서 이런 병렬성을 높이려는 시도들의 명백한 한계점이 보입니다. 다만 primeNumFilter는 전체 실행시간의 95% 정도를 차지하고 있으므로, 암달의 법칙에 따라 전체 실행시간을 최대 20배 가까이 향상시킬 수 있으며 개선이 절실한 상황입니다. 요는 현재 소수를 골라내기 위한 과정에서 반복되는 계산이 많으며, 따라서 절대적인 계산 횟수 혹은 분기 횟수를 대폭 줄일 수 있는 알고리즘으로 교체하는 것입니다. "에라토스테네스의 체"라고 알려진 방법을 도입해 big-O를 획기적으로 줄였습니다. 다음은 source_6입니다.

```
int primeNumFilter(Queue* pq, int A, int B){
    int countPN = 0;
    int sqrtI = (int) mySqrt(B);

    bool* isPrime = (bool *)malloc(sizeof(bool) * (B+1));

    for(int i = 2; i<=B; i++){
        isPrime[i] = true;
    }

    for(int i = 2; i<=sqrtI; i++){
        if (isPrime[i] == false) {
            continue;
        }
        for(int j = i*2; j<= B; j+=i){
            isPrime[j] = false;
        }
    }

    for(int i = A; i<= B; i++){
        if ( isPrime[i] == true){
            enQ(pq, i);
            countPN++;
        }
    }

    free(isPrime);

    return countPN;
}
```

source_6의 실행시간은 다음과 같습니다.



우선 primeNumFilter의 자체 실행시간이 48.30초에서 0.75초로 64배가량, mySqrt의 실행시간이 1.35초에서 0.01초미만으로 100배가량, 전체 실행시간은 49.90초에서 0.98초로 51배 가량 빨라졌습니다. "에라토스테네스의 체" 알고리즘은 2이상의 B이하의 모든 자연수를 저장하는 배열을 생성해야 하므로 많은 memory space를 필요로 한다는 단점이 있지만, 장점이 너무나도 명백합니다. 우선 big-O를 줄일 수 있다는 점도 있지만, source_2~5에서 5천만 회 호출했던 mySqrt를 단 1회만 호출하면 된다는 점도 있습니다. 또한, source_1~5에서는 issue time이 대체로 높으면서 제대로 파이프라인화 되지 않은 것으로 보였던 나누기 연산(%)이 사용되었던 반면, 이 방식에서는 필요 없다는 점도 긍정적으로 작용했으리라고 짐작됩니다. 그러나 이러한 새로운 알고리즘의 도입으로 전체 소요시간에서 primeNumFilter가 차지하는 비율은 source2~5의 95%내외에서 76.64%까지 대폭 줄어들었으나, 여전히 높은 비율임은 특기할만한 사항입니다.

따라서 primeNumFilter에 대한 추가적인 optimization을 시도해야 합니다. source_2에서 source_3으로 변환했을 때처럼 unrolling을 통해 불필요한 분기 문을 줄일 수 있는지 여부를 따져봤습니다. 본 에라토스테네스의 체 알고리즘은 n이 n의 약수에 의해 이미 소수가 아님이 판명되었을 경우 n의 배수 역시 이미 소수가 아니라고 판명되었다는 점을 이용해, 불필요한 계산을 생략하고 있습니다. 여기서 "n이 3이상의 정수일 때 n이 소수라면, $n+1$은 소수가 아니다(n은 홀수, $n+1$짝수이므로)"는 점에서 착안해, source_6에서의 모든 n에 대해 이미 소수로 판명되었는지 여부를 판명하는 if문을 두는 방식에서 n이 이미 소수임이 판명되었다면 n의 배수를 모두 소수가 아니라고 표시한 후 n+1의 계산 없이 바로 $n+2$로 넘어가는 방식으로 변경했습니다.

source_7입니다. 변경된 부분은 다음과 같습니다.

변경 전(source_6)

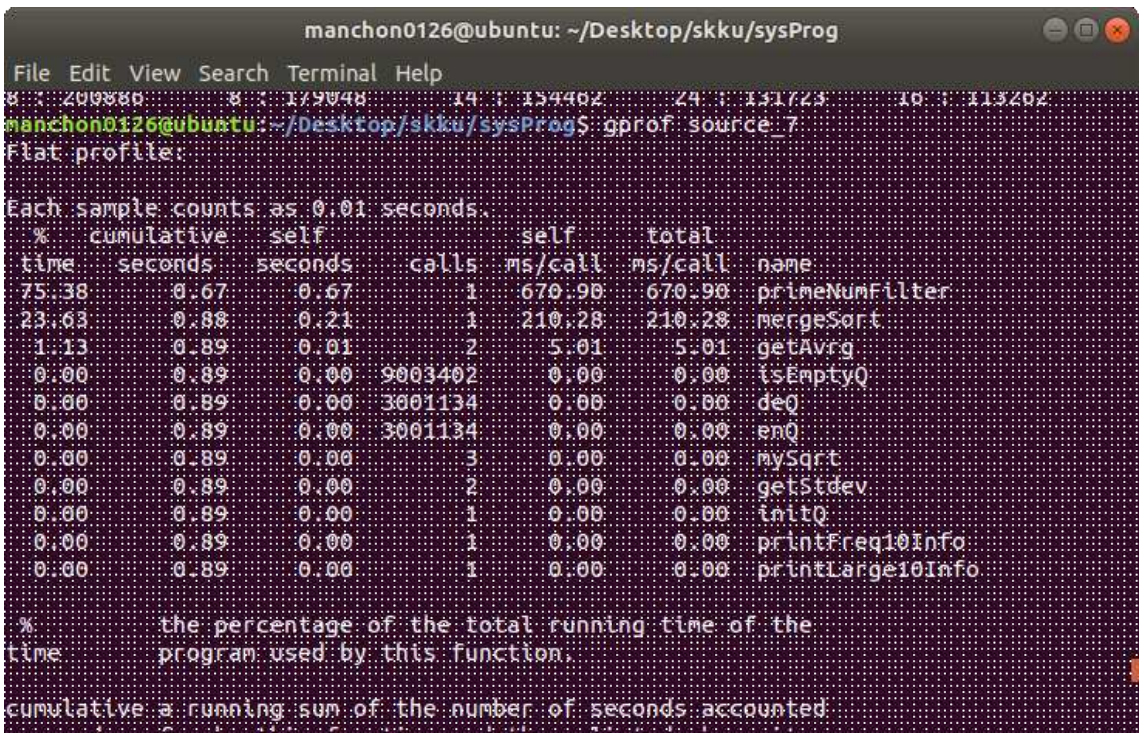
```
int primeNumFilter(Queue* pq, int A, int B){
    int countPN = 0;
    int sqrtI = (int) mySqrt(B);

    bool* isPrime = (bool *)malloc(sizeof(bool) * (B+1));

    for(int i = 2; i<=B; i++){
        isPrime[i] = true;
    }

    for(int i = 2; i<=sqrtI; i++){
        if (isPrime[i] == false) {
            continue;
        }
        for(int j = i*2; j<= B; j+=i){
            isPrime[j] = false;
        }
    }

    for(int i = A; i<= B; i++){
        if ( isPrime[i] == true){
            enQ(pq, i);
            countPN++;
        }
    }

    free(isPrime);

    return countPN;
}
```

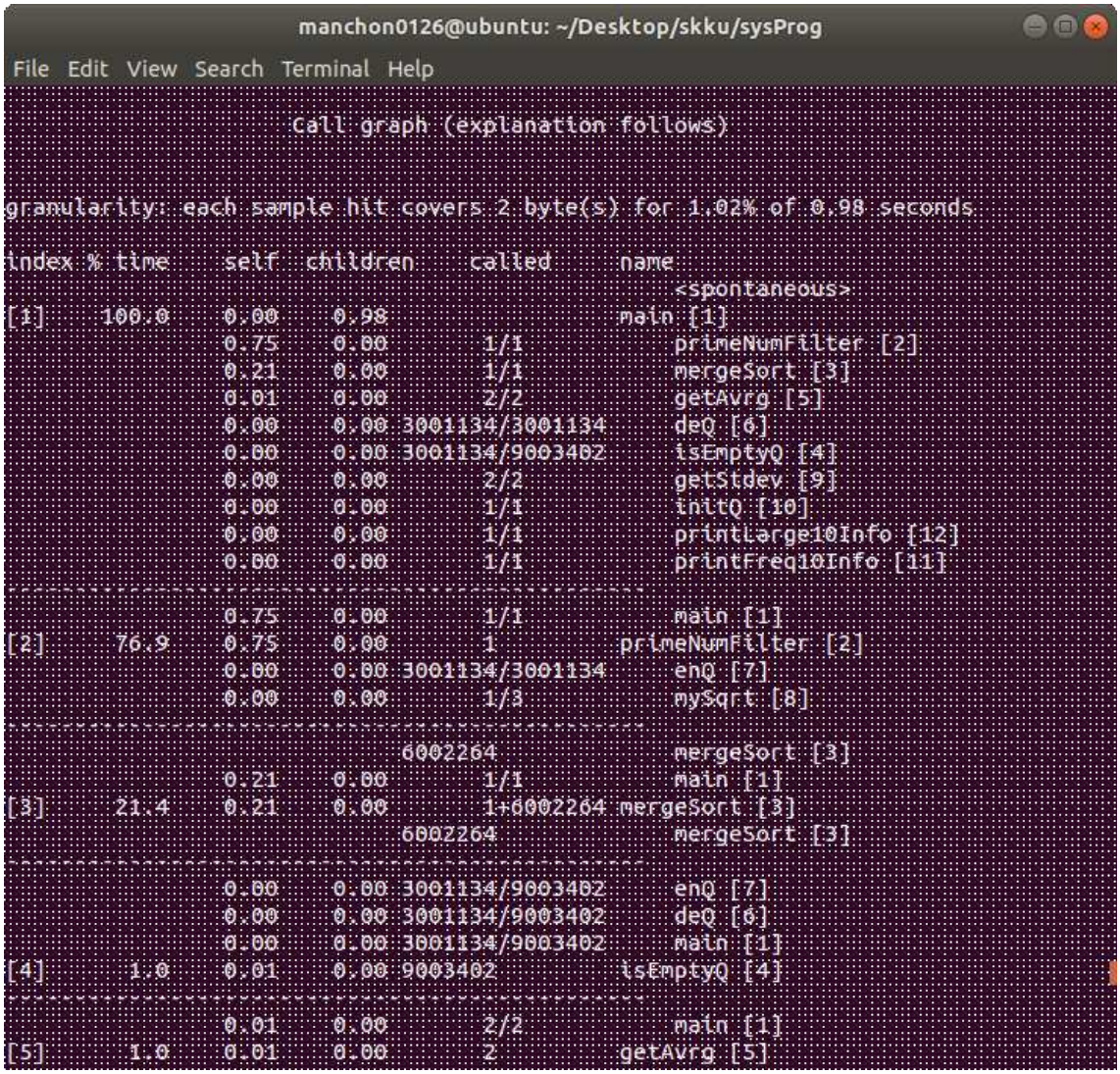
변경 후(source_7)

```
int i0, limit = ((int) mySqrt(B))-1;

(중략)

for(int j = 4; j<= B; j+=2)
    isPrime[j] = false;
```

```
for( i0 = 3 ; i0<=limit; i0+=2){
    int i1 = i0+1;
    if (isPrime[i0] == true) {
        for(int j = i0*2; j<= B; j+=i0)
            isPrime[j] = false;
    }
    else{
        if (isPrime[i1] == true) {
            for(int j = i1*2; j<= B; j+=i1)
                isPrime[j] = false;
        }
    }
}

if( i0 == (limit+1) ){
    if (isPrime[i0]) {
        for(int j = i0*2; j<= B; j+=i0)
            isPrime[j] = false;
    }
}
```

분석결과는 다음과 같습니다.



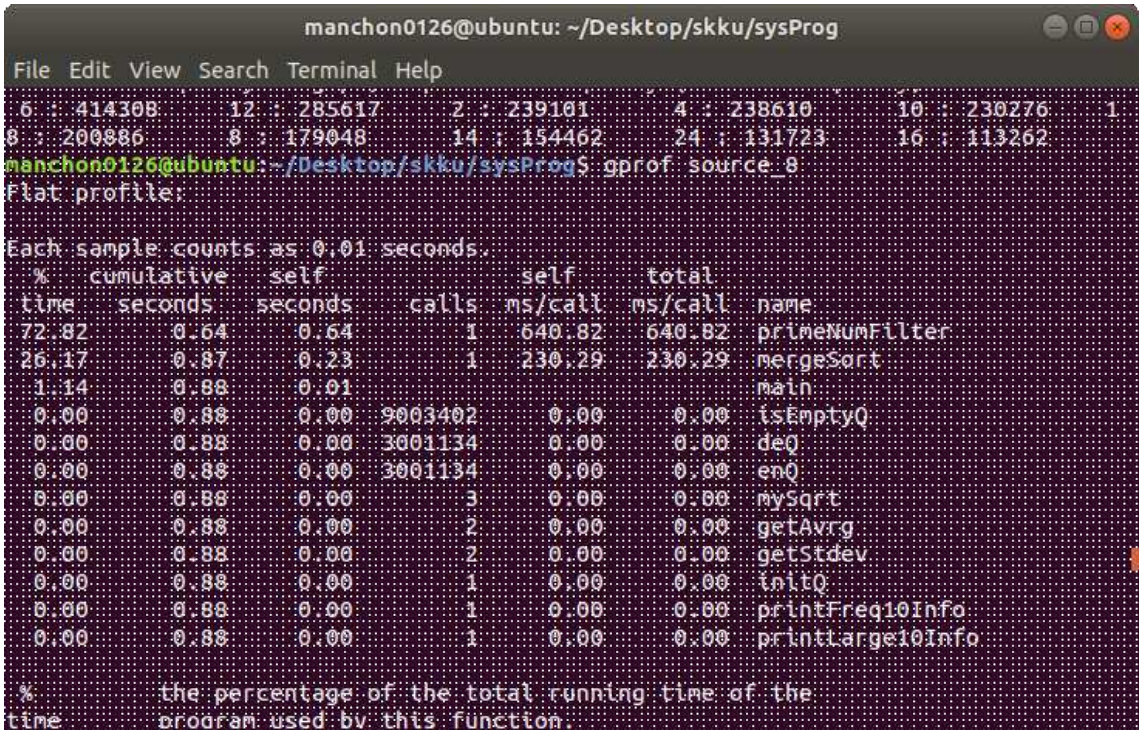


primeNumFilter의 실행시간은 0.75초에서 0.67초로 약 1.15배 향상되었습니다. 상당한 효과를 보였습니다. 따라서 이번에는 조금 더 확장해, "n이 3이상의 정수일 때 n이 소수라면, $n+3$ 역시 소수가 아니다"는 점도 이용하여 4단 unrolling을 시도했습니다. source_8입니다. 변경된 부분은 다음과 같습니다.

```
int i0, limit = ((int) mySqrt(B))-3;

(중략)

for( i0 = 3 ; i0<=limit; i0+=4){
    int i1 = i0+1;
    if (isPrime[i0] == true) {
        int i2 = i0+2;
        for(int j = i0*2; j<= B; j+=i0)
```

```
                isPrime[j] = false;
        if (isPrime[i2] == true) {
            for(int j = i2*2; j<= B; j+=i2)
                isPrime[j] = false;
        }
    }
    else{
        if (isPrime[i1] == true) {
            int i3 = i0+3;
            for(int j = i1*2; j<= B; j+=i1)
                isPrime[j] = false;
            if (isPrime[i3] == true){
                for(int j = i3*2; j<= B; j+=i3)
                    isPrime[j] = false;
            }
        }
        else{
            int i2 = i0+2;
            if (isPrime[i2] == true) {
                for(int j = i2*2; j<= B; j+=i2)
                    isPrime[j] = false;
            }
            else{
                int i3 = i0+3;
                if (isPrime[i3] == true){
                    for(int j = i3*2; j<= B; j+=i3)
                        isPrime[j] = false;
                }
            }
        }
    }
}

for( ; i0<= limit+3; i0++){
    if (isPrime[i0]) {
        for(int j = i0*2; j<= B; j+=i0)
            isPrime[j] = false;
    }
}
```

분석결과는 다음과 같습니다.

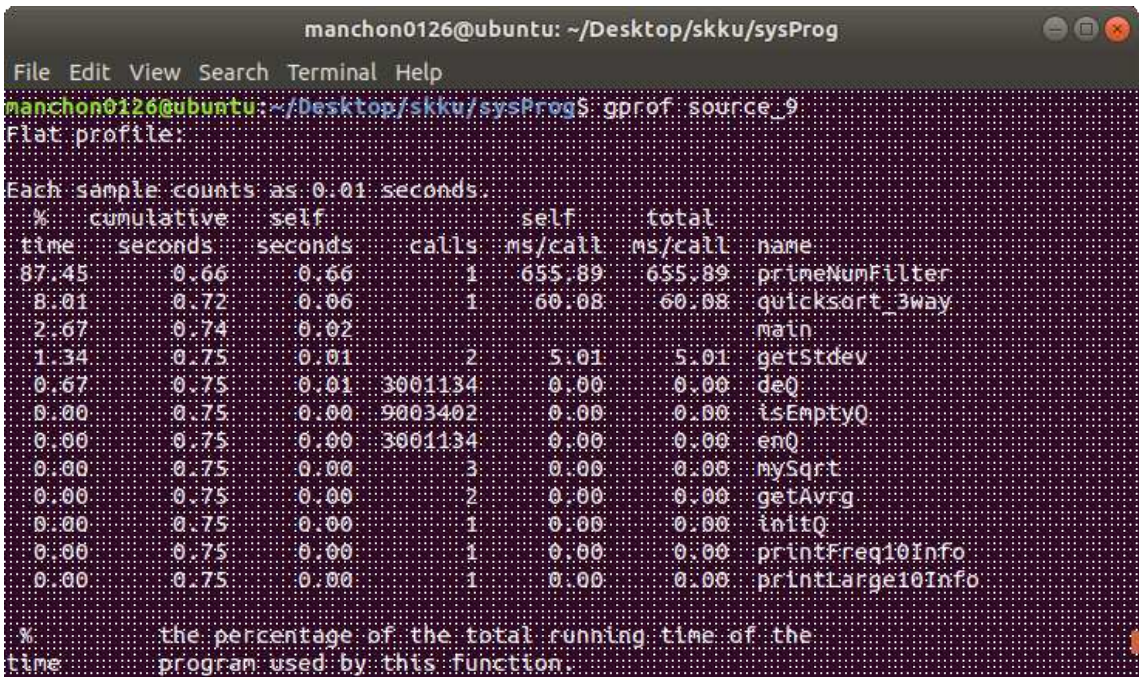


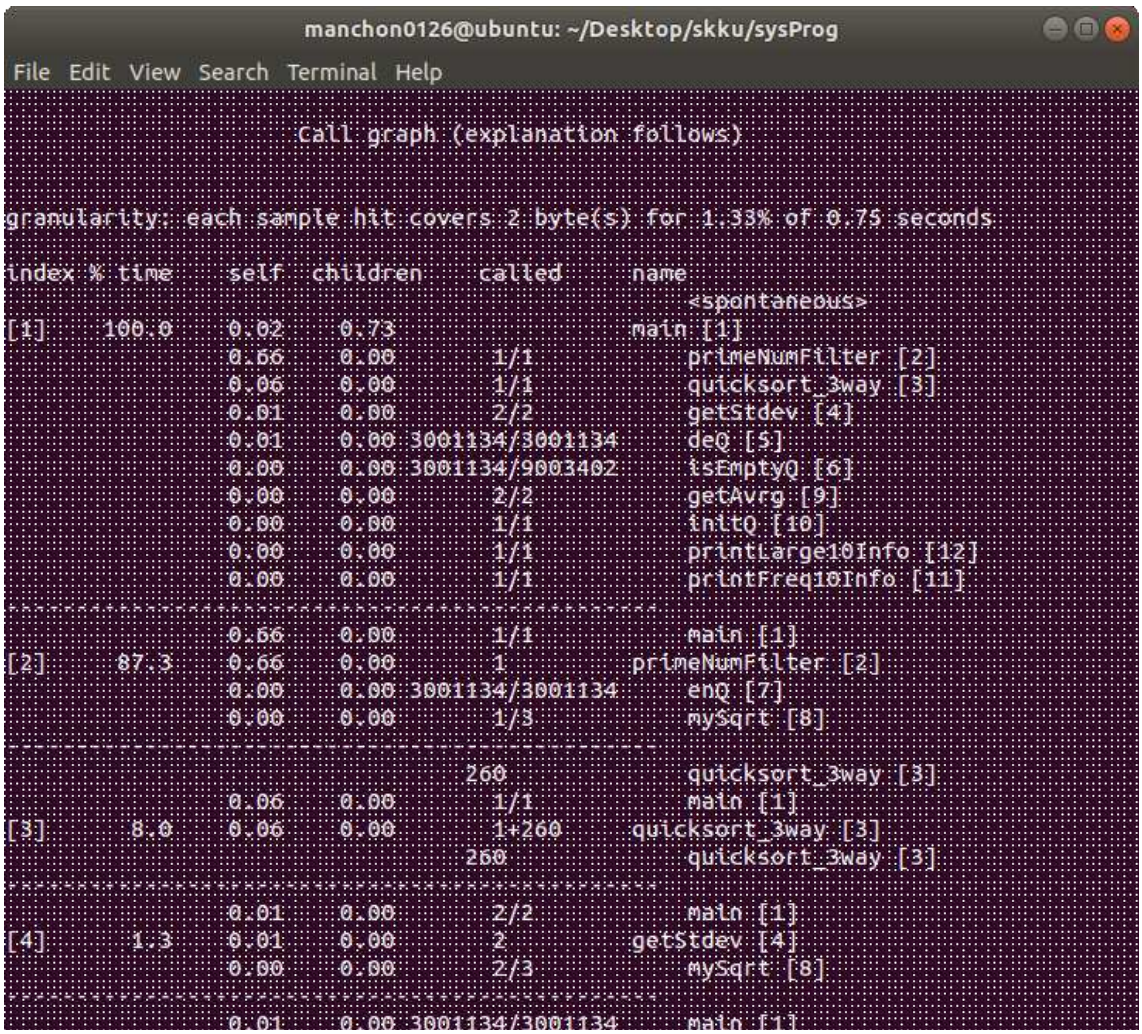
primeNumFilter의 실행시간은 source_7과 비교할 경우 0.67초에서 0.64초로 약 1.04배, source_6과 비교할 경우에는 0.75초에서 0.67초로 약 1.12배 빨라졌습니다. 적지 않은 효과를 보였으나, source_7에서 source_8로 수정할 때 코드의 복잡도가 배 이상 상승했음을 고려할 때, 더 이상 unrolling을 진행하여도 시간 단축 효과에 비해 코드의 가독성과 유지보수의 편리성을 과도하게 해칠 것으로 판단되어 더 이상의 unrolling은 시도하지 않았습니다. 또한, source_3~5에서 보았듯 for문과 if문간의 branch prediction을 통한 병렬성 향상은 이미 최선의 수준일 것이며 추가로 개입할 부분은 대단히 한정적일 것이라 추리할 수 있습니다. 반복적인 enqueue 과정을 분석해도 if문은 더 이상 생략이 불가능하고, 크기가 상위 N%인 소수를 골라내기 위해 차례로 데이터를 삽입하는 queue 구조를 택한 것이므로 병렬적인 enQ함수의 호출 역시 불가능합니다. 여전히 전체 실행시간의 70%가량을 차지하고 있는 primeNumFilter에 대한 추가적인 optimization이 가능하다면 좋을 것이 자명하지만, 한계점에 충분히 가까워졌다고 판단되어 다른 함수들에 대한 optimization을 시도했습니다. 25%가량을 실행시간을 소모하는 정렬 과정에서 mergesort 대신 다른 정렬 알고리즘을 적용해 보기로 했습니다.

우선 quicksort를 시도했습니다. 다만, “소수 사이의 간격”이라는 데이터의 특성상 중복된 값이 대단히 많아 정석적인 quicksort 방식으로는 재귀적 호출이 너무나 많이 필요해 스택 오버플로우에 심각히 취약해진다는 단점이 명백했습니다. 그래서 pivot과 동일한 값은 중앙에 모은 후 재귀 대상에서 제외시키는 변형된 quicksort를("Dutch national flag problem") 사용했습니다. 함수의 이름은 quicksort_3way로 했으며 그 코드는 다음과 같습니다.

```
void quicksort_3way(int* array, int left, int right){
    if(left>=right)
        return;

    else{
        int pivot = array[right];
        int lIdx = left-1, rIdx = right;
        int idx = left;
        int temp;

        while( idx < rIdx){
            if( array[idx] > pivot ){
                lIdx++;
                temp = array[lIdx];
                array[lIdx] = array[idx];
                array[idx] = temp;
                idx++;
            }
            else if( array[idx] < pivot){
                rIdx--;
                temp = array[rIdx];
                array[rIdx] = array[idx];
                array[idx] = temp;
            }
            else
                idx++;
        }

        array[right] = array[lIdx+1];
        array[lIdx+1] = pivot;

        quicksort_3way(array, left, lIdx);
        quicksort_3way(array, rIdx, right);
    }

}
```

source_9의 분석결과입니다.



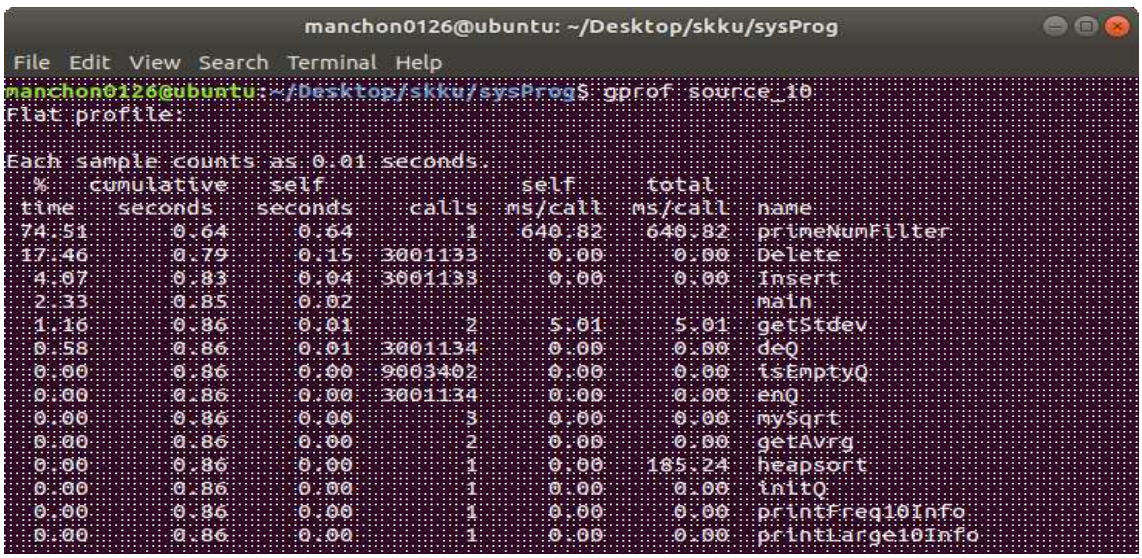


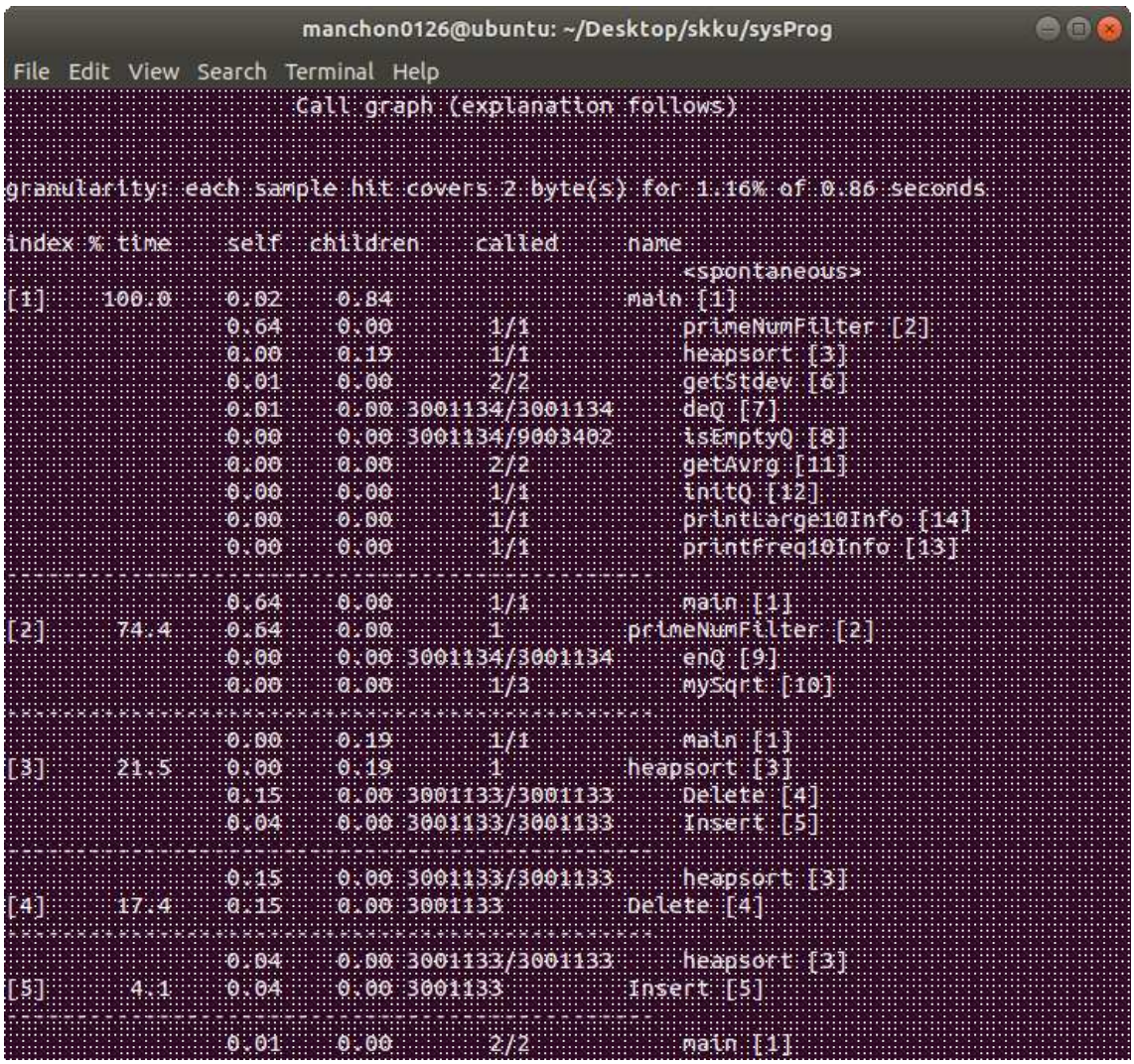
quicksort_3way의 자체적인 실행시간은 0.06초로 source_8에서의 mergesort의 0.23초에 비해 3.8배가량 대폭 향상되었습니다. 다음으로 heapsort를 시도했습니다. 코드는 다음과 같습니다. 다음은 source_10입니다.

```
void heapsort(int a[], int n) {
    Heap heap;

    heap.num = 0;
    heap.items = (int *)malloc( sizeof(int) * (n+1) );

    for (int i = 0; i < n; i++)
        Insert(&heap, a[i]);

    for (int i = 0; i < n; i++)
        a[i] = Delete(&heap);

    free(heap.items);
}

void Insert(Heap *pheap, int data){
    int cur;
    cur = ++(pheap->num);
    while( (cur/2) > 0){
        if( data > pheap->items[cur/2] ) {
            pheap->items[cur] = pheap->items[cur/2];
            cur /= 2;
        }
        else
            break;
    }
    pheap->items[cur] = data;
}
int Delete(Heap *pheap){
    int cur = 1, deeperLV;
    int firstData, lastData;
    firstData = pheap->items[1];
    lastData = pheap->items[pheap->num];
    (pheap->num)--;
    while( cur*2 <= (pheap->num) ){
        if( cur*2 == (pheap->num) ){
```

```
            if( pheap->items[cur*2] > lastData ){
                pheap->items[cur] = pheap->items[cur*2];
                cur *= 2;
            }
            else
                break;
            }
        else{
            if( pheap->items[cur*2] > pheap->items[cur*2+1] ){
                deeperLV = cur*2;
            }
            else{
                deeperLV = cur*2+1;
            }
            if( pheap->items[deeperLV] > lastData){
                pheap->items[cur] = pheap->items[deeperLV];
                cur = deeperLV;
            }
            else{
                break;
            }
        }
    }
    pheap->items[cur] = lastData;
    return firstData;
}
```

실행시간은 다음과 같습니다.





heapsort의 하청 함수들 Delete와 Insert의 실행시간의 합은 0.19초로, mergesort의 0.23초 보다는 1.2배가량 향상되었으나 quicksort_3way의 0.06초 보다는 3배 이상 느립니다. 따라서 quicksort를 적용하기로 결정했습니다.

다음으로 source_7에서 5.01ms의 실행시간을 보였던 getAvrg와, source_9, 10에서 마찬가지로 5.01ms의 실행시간을 보였던 getStdev함수입니다. 사실, source_8에서는 두 함수 모두 0ms의 실행시간을 기록하였고, 다른 3개의 파일에서도 두 개의 함수가 번갈아가며 서로 완벽히 동일한 실행시간을 기록했기 때문에 측정 오차가 매우 강하게 의심되는 부분입니다. 하지만, 이 두 함수는 각각 평균과 표준편차를 구하기 위해 배열 내의 모든 변수에 대해 linear한 연산을 진행한다는 점에서 parallelism을 향상시키기 위한 unrolling이 매우 용이하다는 부분이 있음에 주목했습니다. source_11입니다.

변경 전(source_10)

```
float getAvrg(int* array, int len, int portion){
    int firstIdx = len*(1.0-(float)portion/100);
```

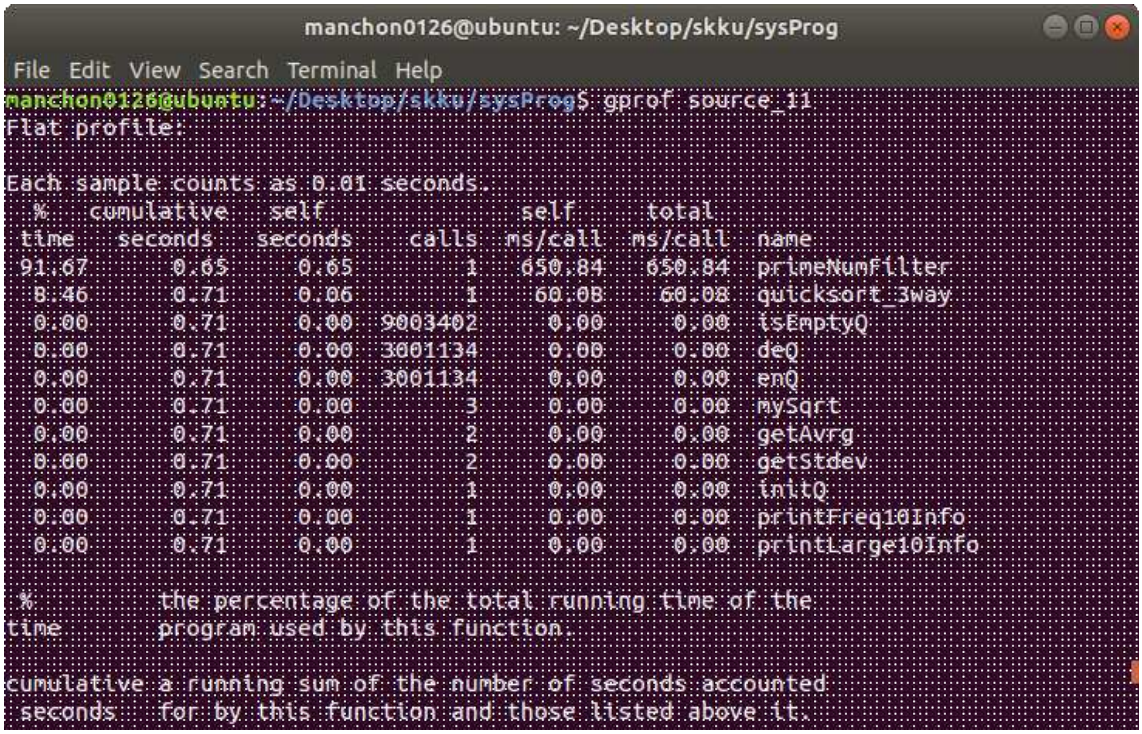
```
    int sum = 0;

    for(int i=(len-1); i>=firstIdx; i--){
        sum += array[i];
    }

    return (float)sum / (len-firstIdx);
}

float getStdev(int* array, int len, float avrg, int portion){
    int firstIdx = len*(1.0-(float)portion/100);
    float sum = 0.0;
    float deviation = 0.0 ;

    for(int i=(len-1); i>=firstIdx; i--){
        deviation = array[i] - avrg;
        sum += deviation*deviation;
    }

    return mySqrt(sum / (len-firstIdx));
}
```

변경 후(source_11)

```
float getAvrg(int* array, int len, int portion){
    int firstIdx = len*(1.0-(float)portion/100);
    int sum = 0, sum2 = 0, sum3 = 0, sum4 = 0, sum5= 0;
    int i = len-1, limit = firstIdx+4;

    for( ; i>=limit; i-=5){
        sum += array[i];
        sum2 += array[i-1];
        sum3 += array[i-2];
        sum4 += array[i-3];
        sum5 += array[i-4];
    }

    for( ; i>=firstIdx; i--){
        sum += array[i];
    }

    sum = sum + sum2 + sum3 + sum4 + sum5;
```

```
    return (float)sum / (len-firstIdx);
}

float getStdev(int* array, int len, float avrg, int portion){
    int firstIdx = len*(1.0-(float)portion/100);
    float sum = 0.0, sum2 = 0.0, sum3 = 0.0, sum4 = 0.0, sum5 = 0.0;
    float deviation, deviation2, deviation3, deviation4, deviation5 ;
    int i = len-1, limit = firstIdx+4;

    for( ; i>=limit; i-=5){
        deviation = array[i] - avrg;
        sum += deviation*deviation;

        deviation2 = array[i+1] - avrg;
        sum2 += deviation*deviation;

        deviation3 = array[i+2] - avrg;
        sum3 += deviation*deviation;

        deviation4 = array[i+3] - avrg;
        sum4 += deviation*deviation;

        deviation5 = array[i+4] - avrg;
        sum5 += deviation*deviation;
    }

    for( ; i>=firstIdx; i--){
        deviation = array[i] - avrg;
        sum += deviation*deviation;
    }

    sum = sum + sum2 + sum3 + sum4 + sum5;

    return mySqrt(sum / (len-firstIdx));
}
```

GPROF 분석 결과의 정확성에 대해 의심되는 부분도 있고, ms단위 미만의 더 정밀한 분석은 불가능하다는 한계에 기인해, unrolling 정도는 교재에서 선정한 실행환경에서는 10단이 최적이었다고 제시된 바 있으므로 그 절반인 5단으로 정했습니다. 실행 결과는 다음과 같습니다.



getAvrg와 getStdev 모두 확실하게 0ms로 기록되었습니다. 마지막으로 isEmptyQ는 비록 실행시간은 0ms로 기록되지만 그 호출 횟수가 9백만 회로 대단히 많으므로 최대한 호출하지 않는 편이 오버헤드를 줄일 수 있을 것으로 판단되어 코드를 수정했습니다. source_12입니다.

변경 전(source_11)

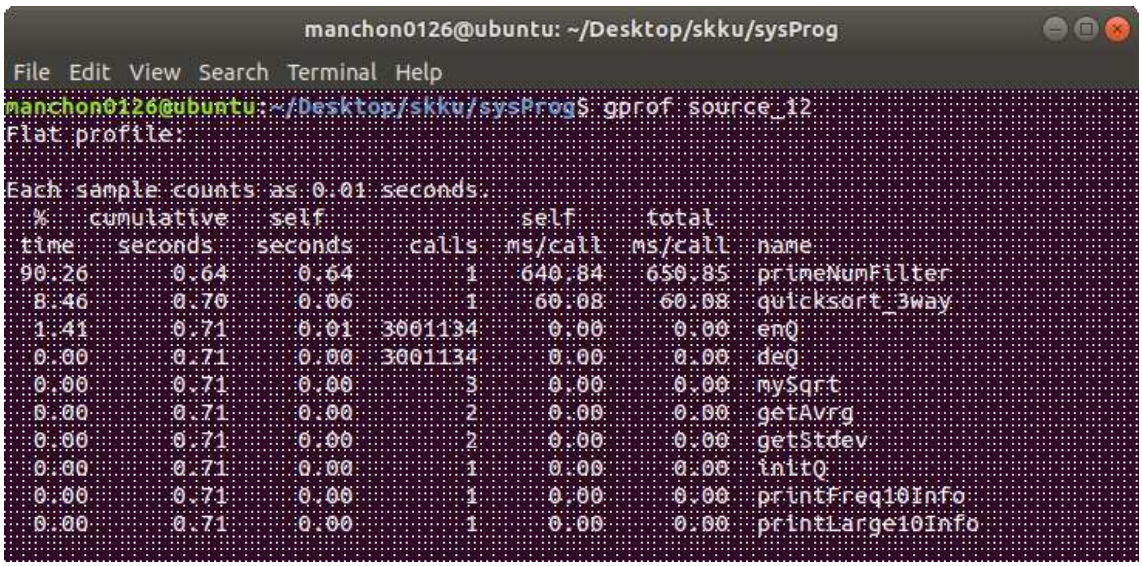
```
void enQ(Queue* pq, Data item){
(중략)
    if(isEmptyQ(pq)){
        pq->front = pq->rear = newNode;
    }
(하략)
}

Data deQ(Queue* pq){
(중략)
    if(isEmptyQ(pq)){
        exit(1);
    }
(하략)
}

void main(void){
```

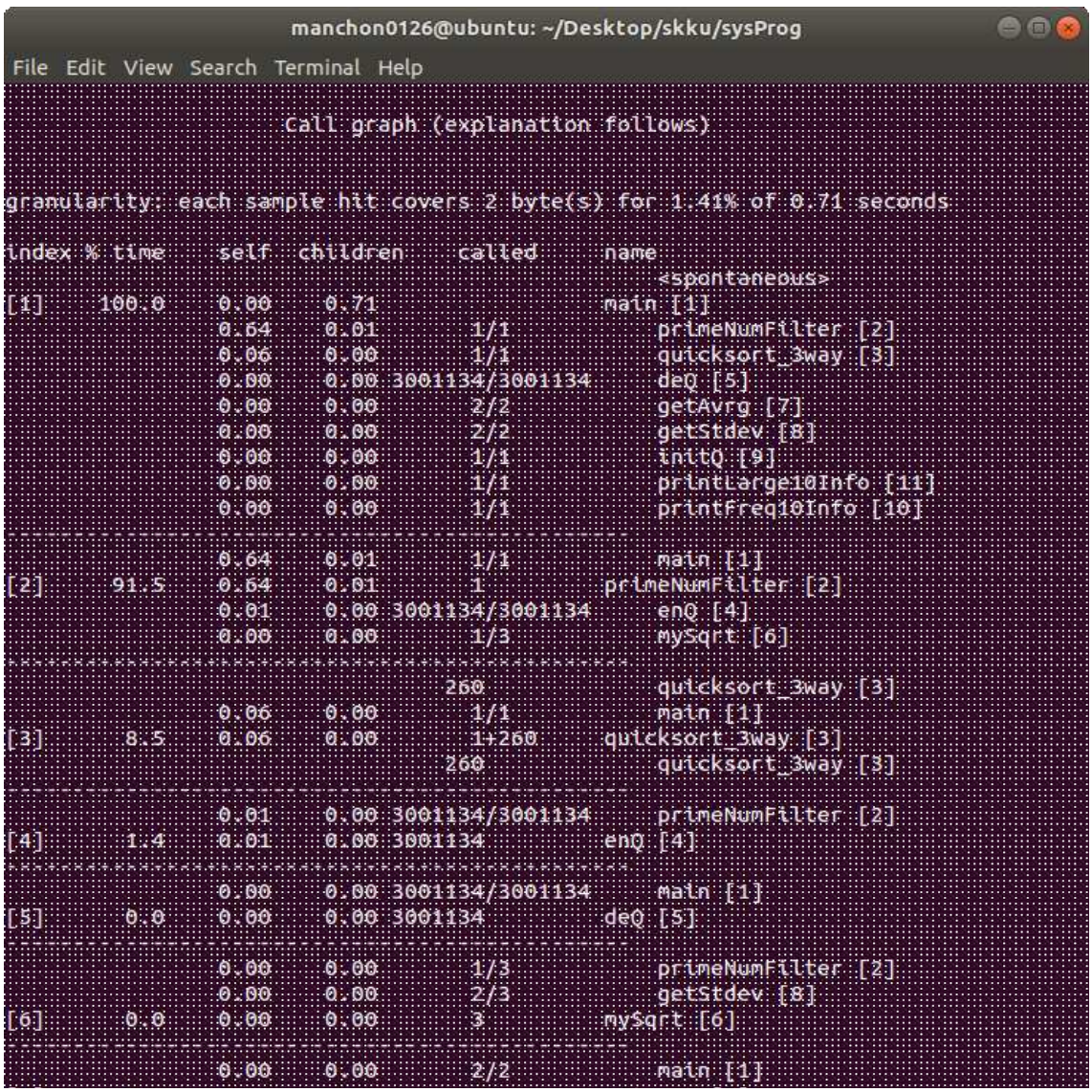
```
(중략)
while( !isEmptyQ(&myQ) ){
    Data temp = deQ(&myQ);
    arrayOfGap[idx] = temp-previousPN;
    previousPN = temp;
    idx++;
}
(하략)
```

변경 후(source_12)

```
void enQ(Queue* pq, Data item){
(중략)
    if( pq->front == NULL ){
        pq->front = pq->rear = newNode;
    }
(하략)
}

Data deQ(Queue* pq){
(중략)
    if( pq->front == NULL) {
        exit(1);
    }
(하략)
}

void main(void){
(중략)
while( !(myQ.front == NULL) ){
    Data temp = deQ(&myQ);
    arrayOfGap[idx] = temp-previousPN;
    previousPN = temp;
    idx++;
}
(하략)
}
```

최종적인 실행 결과는 다음과 같습니다.





source_12로 충분히 optimization이 진행되었다고 판단되어 종료합니다.

4. 최종 결론

총 11회의 optimization이 이루어졌습니다. source_2로 가는 과정에서는 for문의 조건문에 특정 함수가 포함되어 과도하게 호출되고 있다는 점을 인지하여, 해당 함수를 밖으로 빼내었고 해당 함수는 300배 이상, 프로그램 전체는 13배 이상이라는 비약적인 실행시간 감소가 있었습니다. 그 후 주어진 범위 내의 정수 중에서 소수를 분류해내는 "primeNumFilter"가 전체 소요시간에 있어 95%내외의 비중을 차지하므로 해당 함수의 optimization에 집중하였습니다.

source_3~5로 가는 과정에서는 불필요해 보이는 if문의 삭제와 unrolling, 반복적인 enqueue과정에서의 이전 결과 변수와의 계산 생략이라는 3가지 방법을 시도해 병렬성을 높이려 했습니다만, source_3,4 는 이미 prefetching과 branch prediction이 잘 시행되고 있어 효과를 보지 못하거나 오히려 역효과를 보았고, source_5만이 효과가 있었습니다. 다만, source_5역시 "primeNumFilter"가 전체 실행시간의 95%이상을 차지함에도 불구하고 1.05배라는 미소한 소요시간 증가 밖에 보이지 못했습니다.

여기서 더 이상의 병렬성 확보가 힘들 것으로 사료되어, source_6에서는 보다 효율적인 알고리즘을 도입하여, "primeNumFilter"는 64배 정도, 전체 소요시간은 50배 정도 빨라졌습니다.

source_7,8에서는 n이 소수라면 $n+1$, $n+3$은 소수가 아니라는 소수의 성질을 이용하여 불필요한 분기를 줄였습니다. 이러한 원리를 더 확대시켜 적용할 수도 있었으나, 시간단축 효과는 1.12배로 다소 미미함에 반해 코드의 가독성과 유지보수의 편의성을 과도하게 해칠 것으로 판단되어 배제했습니다. 소수를 분석한다는 본 프로그램의 특성상 계산 과정을 줄이기 위해 소정의 분기가 필요함은 자명하고, source_3~8의 사례에서 볼 때에도 추가적인 병렬성을 확보하는 것에도 한계가 있음이 명백하여 "primeNumFilter"에 대한 optimization은 종료하였습니다.

source_9,10에서는 정렬 과정에서 mergesort보다 더 효율적으로 동작하는 정렬 알고리즘을 찾기 위해 quicksort, heapsort를 시도했습니다. 이 때 quicksort는 본 프로그램에 적절히, 그리고 효율적으로 대응하여 동작할 수 있도록 일부 수정했습니다. 결과적으로 수정된 quicksort("quicksort_3way")가 가장 효과가 좋았으며, mergesort대비 3.8배의 속도 향상을 보였습니다.

그 후, source_11에서는 비록 전체 실행시간 중에서는 불과 1% 내외를 차지하고 그나마도 측정 오차로 인해 정확한 측정이 불가능하지만 배열의 전 원소를 linear하게 더하고 곱하므로 unrolling이 매우 용이한 "getAvrg", "getStdev"를 optimize 했습니다.

마지막으로, source_12에서는 정석적인 queue 구현방식에서 enqueue, dequeue과정에서 queue가 비어있는지 확인하는 함수(여기서는 "isEmptyQ")를 호출하지만, 이 호출 과정에서 걸릴 오버헤드를 없애기 위해 함수를 호출하지 않고 확인할 수 있도록 하였습니다.

프로그램의 총소요시간은 500.60초에서 0.71초로 700배 이상 향상되었습니다. 처음에 93.07%의 요소시간을 차지하던 "mySqrt"함수는 465.61초에서 0.01ms이하로 수만 배 이상의 시간 단축 효과가 있었습니다. 다음으로 6.03%를 차지하던 "primeNumFilter"는 초기의 30.18초에서 0.64초로 50배에 가까운 시간 단축이 가능했습니다. 다만, 여전히 전체 소요시간의 90%이상을 차지하는 바 있어 추가적인 optimization이 가능하다면 좋겠지만 정황상 이미 한계라고 판단되어 아쉽지만 끝맺었습니다. 처음 0.03%를 차지했던 정렬 과정은 이후 지분이 최대 26.17%까지 상승했었으나(source_8) 효율적인 정렬 알고리즘의 도입으로 초기의 0.14초에서 0.06초로 2.3배 가량 시간 단축이 가능했습니다. 그 외에 소요시간 비중은 원래 매우 미미했기에 두드러지는 성과를 찾기는 어렵지만 몇몇 함수에 대해 unrolling과 불필요한 함수 호출 방지를 적용함으로써 optimization을 마무리했습니다.