


Supplementary Materials: Source Code of "Evaluation of Strategies for the Development of Efficient Code for Raspberry Pi Devices"

Javier Corral-García , José-Luis González-Sánchez and Miguel-Ángel Pérez-Toledano

1	Contents	
2	Test template used in all experiments	3
3	T1. Bit fields	4
4	T2. Boolean return	4
5	T3. Cascaded function calls	5
6	T4. Row-major accessing	5
7	T5. Constructor initialization lists	6
8	T6. Common subexpression elimination	6
9	T7. Mapping structures	7
10	T8. Dead code elimination	8
11	T9. Exception handling	8
12	T10. Global variables within loops	8
13	T11. Function inlining	9
14	T12. Global variables	9
15	T13. Constants inside loops	10
16	T14. Initialization versus assignment	10
17	T15. Division by a power-of-two denominator	11
18	T16. Multiplication by a power-of-two factor	11
19	T17. Integer versus character	11
20	T18. Loop count down	12
21	T19. Loop unrolling	12
22	T20. Passing structures by reference instead of value	13
23	T21. Pointer aliasing	13

24	T22. Chains of pointers	14
25	T23. Pre-increment versus post-increment	14
26	T24. Linear search	15
27	T25. Invariant IF statements within loops	15

28 Test template used in all experiments

```

30 #include <cmath>
31 #include <iostream>
32 #include "watchtime.h"
33
34 #ifdef level
35     #define OPTIMIZE __attribute__((optimize(level)))
36 #else
37     #define OPTIMIZE
38 #endif
39
40 const int nms = 10;           // number of measurements
41 const int reps = 100000000; // number of times a test is repeated (for a single measurement)
42
43 void stats(unsigned times[]){
44     unsigned totaltime, meantime, sumsq, sd;
45     float testtime;
46     totaltime = meantime = sumsq = 0;
47     printf("\t\t\t_____\\n");
48     for (int i=0; i<nms; i++){
49         printf("\t\tTime_%d\\t%lu_\\n", i+1, times[i]);
50         totaltime += times[i];
51     }
52     meantime = totaltime / nms;
53     for (int i=0; i<nms; i++){
54         sumsq += (times[i] - meantime) * (times[i] - meantime);
55     }
56     sd = sqrt(sumsq / (nms-1));
57     testtime = ((float)meantime / (float)reps);
58     testtime *= 1000000; // ms to ns
59     printf("\t\t\t_____\\n");
60     printf("\t\tAverage_measurement_time:_%lu_\\n", meantime);
61     printf("\t\tStandard_deviation:_%lu_\\n", sd);
62     printf("\t\t\t_____\\n");
63     printf("\t\tAverage_test_time:_%.2f_ns\\n\\n", testtime);
64 }
65
66
67 int main() {
68     int i, j;
69     unsigned times[nms], aux;
70     Watchtime time;
71
72     printf("\\n\\nTechnique_1:_Bit_fields\\n\\n");
73     printf("\\tOptimization_level_%s\\n\\n", level);
74
75     printf("\\tTest_1_(standard_code)...\\n\\n");
76     for (i=0; i<nms; i++){
77         time.startTime();
78         for (j=0; j<reps; j++){
79             test1();
80         }
81         aux = time.getTime();
82         times[i] = aux;
83     }
84     stats(times);
85
86     printf("\\tTest_2_(efficient_code)...\\n\\n");
87     for (i=0; i<nms; i++){
88         time.startTime();
89         for (j=0; j<reps; j++){
90             test2();
91         }
92         aux = time.getTime();
93         times[i] = aux;
94     }
95     stats(times);
96 }

```

98 T1. Bit fields

```

99
100 typedef struct {
101     unsigned int bitA : 1;
102     unsigned int bitB : 1;
103     unsigned int bitC : 1;
104     unsigned int bitD : 1;
105 } BitField;
106
107 typedef struct {
108     unsigned int bits;
109 } IntegerBitField;
110
111 unsigned int OPTIMIZE getBitField(const BitField *d) {
112     return (d->bitA << 0) |
113           (d->bitB << 1) |
114           (d->bitC << 2) |
115           (d->bitD << 3);
116 }
117
118 unsigned int OPTIMIZE getIntegerBitField(const IntegerBitField *d) {
119     return d->bits;
120 }
121
122 unsigned int test1() {
123     BitField *p;
124     return getBitField(p);
125 }
126
127 unsigned int test2() {
128     IntegerBitField *p;
129     return getIntegerBitField(p);
130 }

```

132 T2. Boolean return

```

133
134 bool OPTIMIZE getOR(bool argA, bool argB, bool argC, bool argD) {
135     return (argA || argB || argC || argD);
136 }
137
138 typedef unsigned int Flags;
139
140 #define flagA (1u << 0)
141 #define flagB (1u << 1)
142 #define flagC (1u << 2)
143 #define flagD (1u << 3)
144
145 bool OPTIMIZE getFlagsOR(Flags flags) {
146     return (flags & (flagA | flagB | flagC | flagD)) != 0;
147 }
148
149 void test1() {
150     getOR(1,1,0,0);
151 }
152
153 void test2() {
154     getFlagsOR(1100);
155 }

```

T3. Cascaded function calls

```

158 int const N = 20;
159 int a[N];
160
161 class Class {
162     private:
163         int data;
164     public:
165         void setData(int data);
166         int  getData();
167 };
168
169 void Class::setData(int data) {
170     Class::data = data;
171 }
172
173 int Class::getData(){
174     return data;
175 }
176
177 void OPTIMIZE test1(Class *c){
178     for (int i=0; i<N; i++){
179         if (c->getData()==1){
180             a[i] = 0;
181         }
182     }
183 }
184
185 void OPTIMIZE test2(Class *c){
186     int data = c->getData();
187     for (int i=0; i<N; i++){
188         if (data==1){
189             a[i] = 0;
190         }
191     }
192 }
193

```

T4. Row-major accessing

```

196 int const N = 60;
197 int array[N][N];
198
199 void OPTIMIZE test1(){
200     for (int j=0; j<N; j++)
201         for (int i=0; i<N; i++)
202             array[i][j] = 0;
203 }
204
205 void OPTIMIZE test2(){
206     for (int i=0; i<N; i++)
207         for (int j=0; j<N; j++)
208             array[i][j] = 0;
209 }
210

```

T5. Constructor initialization lists

```

using namespace std;

class ClassA {
private:
    string dataA;
    string dataB;
    int dataC;
    int dataD;
public:
    ClassA(string data1, string data2, int data3, int data4);
};

OPTIMIZE ClassA::ClassA(string data1, string data2, int data3, int data4) {
    dataA = data1;
    dataB = data2;
    dataC = data3;
    dataD = data4;
}

class ClassB {
private:
    string dataA;
    string dataB;
    int dataC;
    int dataD;
public:
    ClassB(string data1, string data2, int data3, int data4);
};

OPTIMIZE ClassB::ClassB(string data1, string data2, int data3, int data4):
    dataA(data1), dataB(data2), dataC(data3), dataD(data4) {
}

void OPTIMIZE test1(){
    ClassA c("data1","data2",1,1);
}

void OPTIMIZE test2(){
    ClassB c("data1","data2",1,1);
}

```

T6. Common subexpression elimination

```

int x=100;
int i,j;

void OPTIMIZE test1(){
    i = x + sqrt(16384) + 1;
    j = x + sqrt(16384);
}

void OPTIMIZE test2(){
    int aux = x + sqrt(16384);
    i = aux + 1;
    j = aux;
}

```

T7. Mapping structures

```

272 #define NELEMS(a) ((int) (sizeof(a) / sizeof(a[0])))
273
274
275 static const struct {
276     const char data[7]; /* NB. PIC */
277     int value;
278 } map[] = {
279     { "AAAAAA", 1 },
280     { "BBBBBB", 2 },
281     { "CCCCCC", 3 },
282     { "DDDDDD", 4 },
283     { "EEEEEE", 5 },
284     { "FFFFFF", 6 },
285     { "GGGGGG", 7 },
286     { "HHHHHH", 8 },
287     { "IIIIII", 9 },
288     { "JJJJJJ", 9 },
289     { "KKKKKK", 9 },
290     { "LLLLLL", 9 },
291     { "MMMMMM", 9 },
292     { "OOOOOO", 9 },
293     { "PPPPPP", 9 },
294     { "QQQQQQ", 9 },
295     { "RRRRRR", 9 },
296     { "SSSSSS", 9 },
297     { "TTTTTT", 9 }
298 };
299
300 int OPTIMIZE dataToValue(const char *data) {
301     if (strcmp(data, "AAAAAA") == 0) return 1;
302     else if (strcmp(data, "BBBBBB") == 0) return 2;
303     else if (strcmp(data, "CCCCCC") == 0) return 3;
304     else if (strcmp(data, "DDDDDD") == 0) return 4;
305     else if (strcmp(data, "EEEEEE") == 0) return 5;
306     else if (strcmp(data, "FFFFFF") == 0) return 6;
307     else if (strcmp(data, "GGGGGG") == 0) return 7;
308     else if (strcmp(data, "HHHHHH") == 0) return 8;
309     else if (strcmp(data, "IIIIII") == 0) return 9;
310     else if (strcmp(data, "JJJJJJ") == 0) return 10;
311     else if (strcmp(data, "KKKKKK") == 0) return 11;
312     else if (strcmp(data, "LLLLLL") == 0) return 12;
313     else if (strcmp(data, "MMMMMM") == 0) return 13;
314     else if (strcmp(data, "NNNNNN") == 0) return 14;
315     else if (strcmp(data, "OOOOOO") == 0) return 15;
316     else if (strcmp(data, "PPPPPP") == 0) return 16;
317     else if (strcmp(data, "QQQQQQ") == 0) return 17;
318     else if (strcmp(data, "RRRRRR") == 0) return 18;
319     else if (strcmp(data, "SSSSSS") == 0) return 19;
320     else if (strcmp(data, "TTTTTT") == 0) return 20;
321     else return -1; /* default case */
322 }
323
324 int OPTIMIZE dataToValue2(const char *data) {
325     for (int i = 0; i < NELEMS(map); i++)
326         if (strcmp(data, map[i].data) == 0)
327             return map[i].value;
328     return -1; // default case
329 }
330
331 int test1() {
332     return dataToValue("TTTTTT");
333 }
334
335 int test2() {
336     return dataToValue2("TTTTTT");
337 }

```

T8. Dead code elimination

```

340 int global;
341
342
343 void OPTIMIZE test1(){
344     int i;
345     i = 1;           // dead store
346     global = 1;      // dead store
347     global = 2;
348     return;
349     global = 3;      // unreachable
350 }
351
352 void OPTIMIZE test2(){
353     global = 2;
354     return;
355 }

```

T9. Exception handling

```

358 using namespace std;
359
360
361 class myexception: public exception {
362 } myex;
363
364 int OPTIMIZE test1(){
365     int num = 100;
366     for (int i=0; i<1; i++){
367         try{
368             if (num == 100) {
369                 throw myex;
370             }
371         } catch (exception& e){
372         }
373     }
374     return 0;
375 }
376
377 int OPTIMIZE test2(){
378     int num = 100;
379     for (int i=0; i<1; i++){
380         if (num != 100) {
381             continue;
382         }
383     }
384     return 0;
385 }

```

T10. Global variables within loops

```

388 int const N = 20;
389 int a[N];
390 int sum;
391
392
393 void initializeArray(int N){
394     for (int i=0; i<N; i++){
395         a[i] = i;
396     }
397 }
398
399 void OPTIMIZE test1(){
400     sum = 0;
401     for (int i=0; i<N; i++)
402         sum += a[i];
403 }
404
405 void OPTIMIZE test2(){
406     int t = 0;
407     for (int i=0; i<N; i++)
408         t += a[i];
409     sum = t;
410 }

```


T11. Function inlining

```
int OPTIMIZE add (int x, int y) {  
    return x + y;  
}  
  
int OPTIMIZE sub(int x, int y) {  
    return add (x, -y);  
}  
  
int OPTIMIZE sub2(int x, int y) {  
    return x + -y;  
}  
  
void test1(){  
    sub(10,5);  
}  
  
void test2(){  
    sub2(10,5);  
}
```

T12. Global variables

```
int value;  
  
int f() {  
    return 512;  
}  
  
void OPTIMIZE test1() {  
    for (int i=0; i<50; i++) {  
        value += f();  
    }  
}  
  
void OPTIMIZE test2() {  
    int aux = value;  
    for (int i=0; i<50; i++) {  
        aux += f();  
    }  
    value = aux;  
}
```

T13. Constants inside loops

```

456
457
458 #define IDENTIFIER_A 2
459 #define IDENTIFIER_B 1
460 #define IDENTIFIER_C 3
461
462 typedef struct {
463     unsigned int value;
464 } Structure;
465
466 void aux1(int i){
467     i++;
468 }
469 void aux2(int i){
470     i++;
471 }
472 void aux3(int i){
473     i++;
474 }
475
476
477 void OPTIMIZE test1(int N) {
478     int i;
479     Structure t,*pt;
480     pt = &t;
481
482     pt->value &= IDENTIFIER_C;
483
484     for (i=0; i<N; i++) {
485         if (pt->value & IDENTIFIER_A)
486             aux1(i);
487         else if (pt->value & IDENTIFIER_B)
488             aux2(i);
489         else
490             aux3(i);
491     }
492 }
493
494 void OPTIMIZE test2(int N) {
495     int i;
496     Structure t,*pt;
497     pt = &t;
498
499     pt->value &= IDENTIFIER_C;
500
501     if (pt->value & IDENTIFIER_A) {
502         for (i=0; i<N; i++)
503             aux1(i);
504     } else if (pt->value & IDENTIFIER_B) {
505         for (i=0; i<N; i++)
506             aux2(i);
507     } else {
508         for (i=0; i<N; i++)
509             aux3(i);
510     }
511 }
512

```

T14. Initialization versus assignment

```

513
514
515 void OPTIMIZE test1() {
516     std::complex<double> mycomplex;
517     mycomplex = (3.14);
518 }
519
520 void OPTIMIZE test2() {
521     std::complex<double> mycomplex(3.14);
522 }
523

```

524 T15. Division by a power-of-two denominator

```

525
526 int OPTIMIZE divide (unsigned int i) {
527     return i / 1024;
528 }
529
530 int OPTIMIZE divide2 (unsigned int i) {
531     return i >> 10;
532 }
533
534 void test1(){
535     divide(100000000);
536 }
537
538 void test2(){
539     divide2(100000000);
540 }

```

542 T16. Multiplication by a power-of-two factor

```

543
544 int OPTIMIZE multiply (int i) {
545     return i * 1024;
546 }
547
548 int OPTIMIZE multiply2 (unsigned int i) {
549     return i >> 10;
550 }
551
552 void test1(){
553     multiply(100000000);
554 }
555
556 void test2(){
557     multiply2(100000000);
558 }

```

560 T17. Integer versus character

```

561
562 char OPTIMIZE sum_char(char a, char b, char c, char d, char e) {
563     return a+b+c+d+e;
564 }
565
566 int OPTIMIZE sum_int(int a, int b, int c, int d, int e) {
567     return a+b+c+d+e;
568 }
569
570 void test1(){
571     sum_char(1,2,3,4,5);
572 }
573
574 void test2(){
575     sum_int(1,2,3,4,5);
576 }
577

```

T18. Loop count down

```

578
579
580 int const N = 100;
581 int a[N];
582
583 void OPTIMIZE test1(){
584     for (int i=0; i<N; i++) {
585         a[i]=i;
586     }
587 }
588
589 void OPTIMIZE test2(){
590     int i = N+1;
591     while (--i) {
592         a[i] = i;
593     }
594 }

```

T19. Loop unrolling

```

596
597
598 const int N = 50;
599 int array[N];
600
601 void OPTIMIZE initialization1(){
602     int i;
603     for (i=0; i<N; i++){
604         array[i] = 0;
605     }
606 }
607
608 void OPTIMIZE initialization2(){
609     int i;
610     for (i=0; i<N; i+=5){
611         array[i] = 0;
612         array[i+1] = 0;
613         array[i+2] = 0;
614         array[i+3] = 0;
615         array[i+4] = 0;
616     }
617 }
618
619 void test1(){
620     initialization1();
621 }
622
623 void test2(){
624     initialization2();
625 }

```

T20. Passing structures by reference instead of value

```

628 using namespace std;
629
630
631 typedef struct {
632     int array[10];
633     int value;
634 } Structure;
635
636 class Class {
637     private:
638         string data_a ;
639         string data_b;
640         Structure structure;
641     public:
642         Class(string data1, string data2, int i);
643         int getIndex();
644 };
645
646 OPTIMIZE Class::Class(string data1, string data2, int i) {
647     data_a = data1;
648     data_b = data2;
649     structure.value = i;
650 }
651
652 int OPTIMIZE Class::getIndex() {
653     return structure.value;
654 }
655
656 int OPTIMIZE test1(Class value){
657     return value.getIndex();
658 }
659
660 int OPTIMIZE test2(Class *reference){
661     return reference->getIndex();
662 }

```

T21. Pointer aliasing

```

665 int a,b,c,d,e;
666 int *pa = &a;
667 int *pb = &b;
668 int *pc = &c;
669 int *pd = &d;
670 int *pe = &e;
671
672
673 void OPTIMIZE pointersA(int *t1, int *t2, int *t3, int *t4, int *step) {
674     *t1 += *step;
675     *t2 += *step;
676     *t3 += *step;
677     *t4 += *step;
678 }
679
680 void OPTIMIZE pointersB(int *t1, int *t2, int *t3, int *t4, int *step) {
681     int s = *step;
682     *t1 += s;
683     *t2 += s;
684     *t3 += s;
685     *t4 += s;
686 }
687
688 void test1() {
689     pointersA(pa,pb,pc,pd,pe);
690 }
691
692 void test2() {
693     pointersB(pa,pb,pc,pd,pe);
694 }

```

T22. Chains of pointers

```

697 typedef struct { int a,b,c,d,e; } Values;
698 typedef struct { Values *values; } Structure;
699
700 Structure structure,*pstructure;
701 Values values;
702
703 void OPTIMIZE test1() {
704     structure.values = &values;
705     pstructure = &structure;
706
707     pstructure->values->a = 0;
708     pstructure->values->b = 0;
709     pstructure->values->c = 0;
710     pstructure->values->d = 0;
711     pstructure->values->e = 0;
712 }
713
714 void OPTIMIZE test2() {
715     structure.values = &values;
716     pstructure = &structure;
717
718     Values *aux = pstructure->values;
719     aux->a = 0;
720     aux->b = 0;
721     aux->c = 0;
722     aux->d = 0;
723     aux->e = 0;
724 }
725

```

T23. Pre-increment versus post-increment

```

728 int const N = 200;
729 int array[N+1];
730
731 void OPTIMIZE test1(){
732     for (int i=0; i<N;){
733         array[i] = i++;
734     }
735 }
736
737 void OPTIMIZE test2(){
738     for (int i=0; i<N;){
739         array[i] = ++i;
740     }
741 }
742

```

744 T24. Linear search

```

745 int const N = 100;
746 int list [N];
747 int *plist;
748
749 void OPTIMIZE inicialize(int *list , int N){
750     for (int i=0; i<N; i++){
751         list[i] = i;
752     }
753 }
754
755 int OPTIMIZE search1(int *list , int N, int want) {
756     int i;
757     for (i = 0; i < N; i++)
758         if (list[i] == want)
759             return i;
760     return -1;
761 }
762
763 int OPTIMIZE search2(int *list , int N, int want) {
764     int i;
765     list[N] = want;
766     i = 0;
767     while (list[i] != want)
768         i++;
769     if (i == N)
770         return -1;
771     return i;
772 }
773
774 int test1(){
775     plist = list;
776     return search1(plist ,N,98);
777 }
778
779 int test2(){
780     plist = list;
781     return search2(plist ,N,98);
782 }
783
784

```

785 T25. Invariant IF statements within loops

```

786 int x=0;
787
788 int const N = 100;
789 int a[N];
790 int b[N];
791
792 void OPTIMIZE test1(){
793     for (int i=0; i<N; i++)
794         if (x==1)
795             a[i] = 0;
796         else
797             b[i] = 0;
798 }
799
800 void OPTIMIZE test2(){
801     if (x==1)
802         for (int i=0; i<N; i++)
803             a[i] = 0;
804     else
805         for (int i=0; i<N; i++)
806             b[i] = 0;
807 }
808
809

```