Decreasing Process Memory Requirements by Overlapping Program Portions

by

Richard Bowman

Emily Ratliff

David Whalley

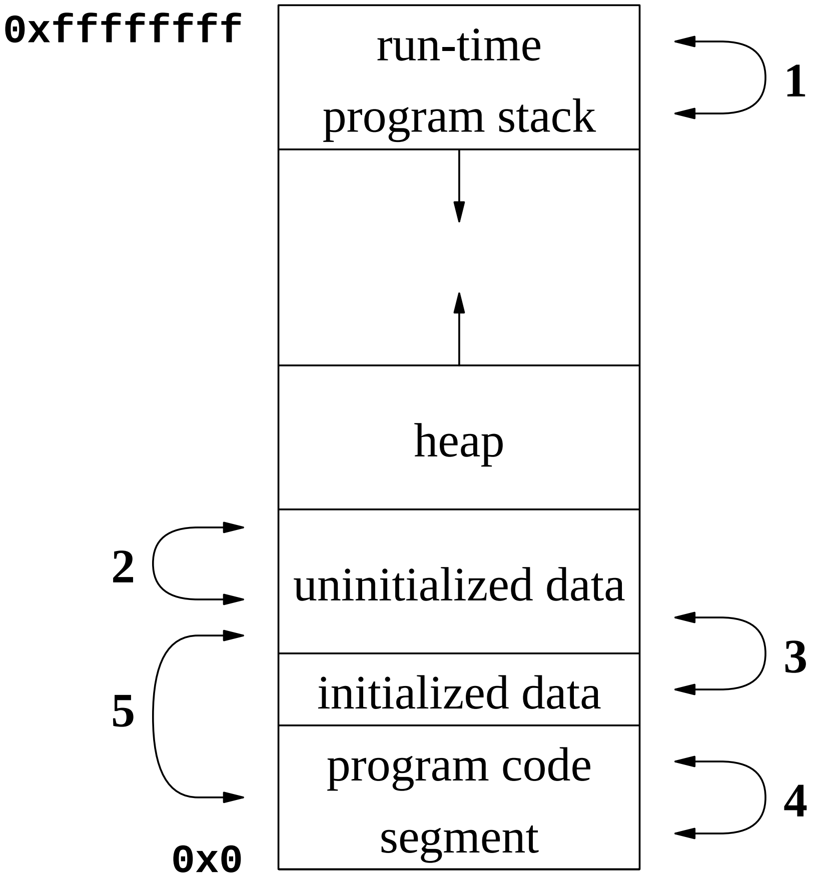
Computer Science Department

Florida State University

Motivation for Decreasing Process Memory Requirements

- May allow embedded systems to meet their strict limitations on program size.
- May improve memory hierarchy performance.
  - — reduce cache misses
  - — reduce page faults
- May help offset increases in code size due to code increasing compiler transformations.
- Automatic overlapping supports the software engineering principle of using descriptive variable names.

Areas for Overlapping Program Portions



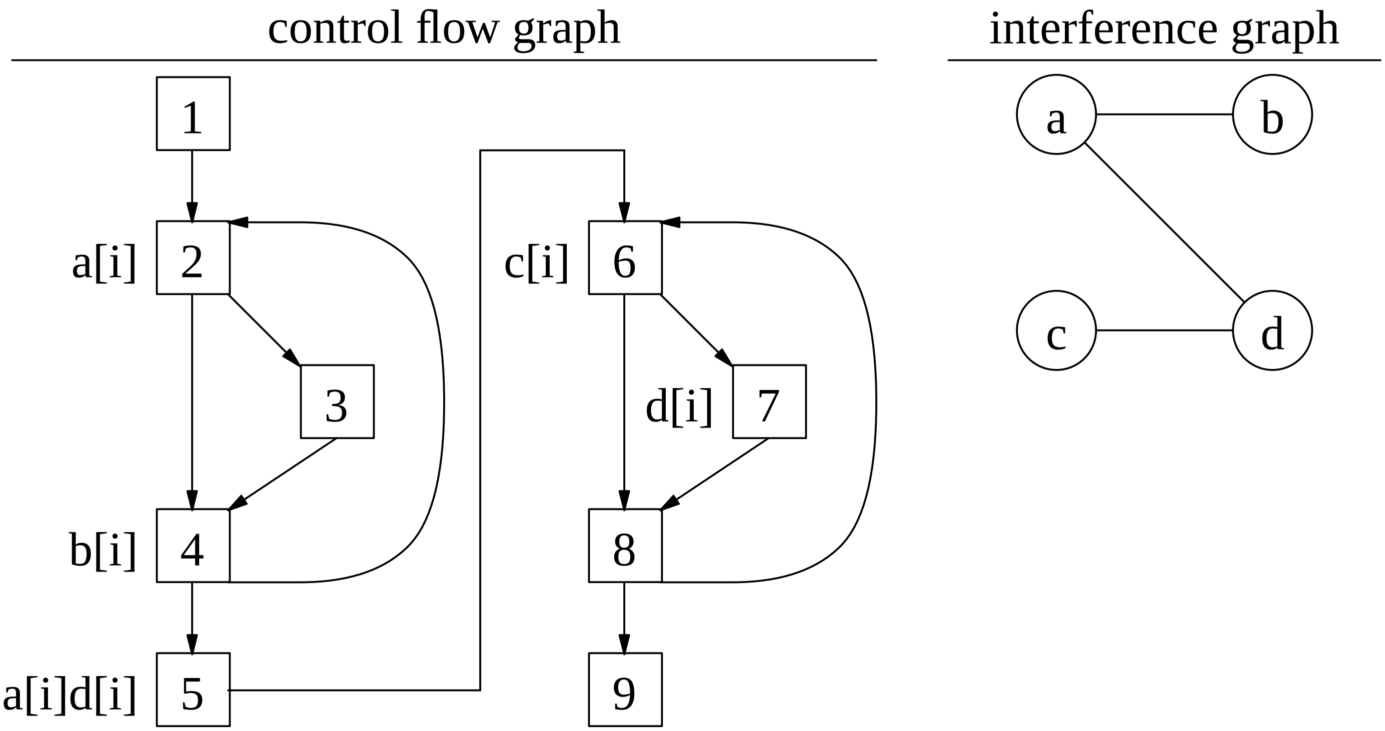
1. **overlap run-time stack data**
2. **overlap uninitialized static data**
3. **overlap uninitialized static data and initialized static data**
4. **overlap instructions**
5. **overlap uninitialized static data and instructions**

Overlapping Data

- Used a graph coloring approach to detect conflicting live ranges.
- Issues
  - Detecting indirectly referenced live ranges.
  - Detecting live ranges of static data used in more than one function.
  - Assigning memory locations to live ranges.

Indirectly Referenced Live Ranges

- Indirectly referenced variables are treated as having a single live range.
- Interference graph nodes not directly connected can be overlapped in memory.



live range = possible predecessors ∩ possible successors

live range of a[] = [1,2,3,4,5] ∩ [2,3,4,5,6,7,8,9] = [2,3,4,5]

live range of b[] = [1,2,3,4] ∩ [2,3,4,5,6,7,8,9] = [2,3,4]

live range of c[] = [1,2,3,4,5,6,7,8] ∩ [6,7,8,9] = [6,7,8]

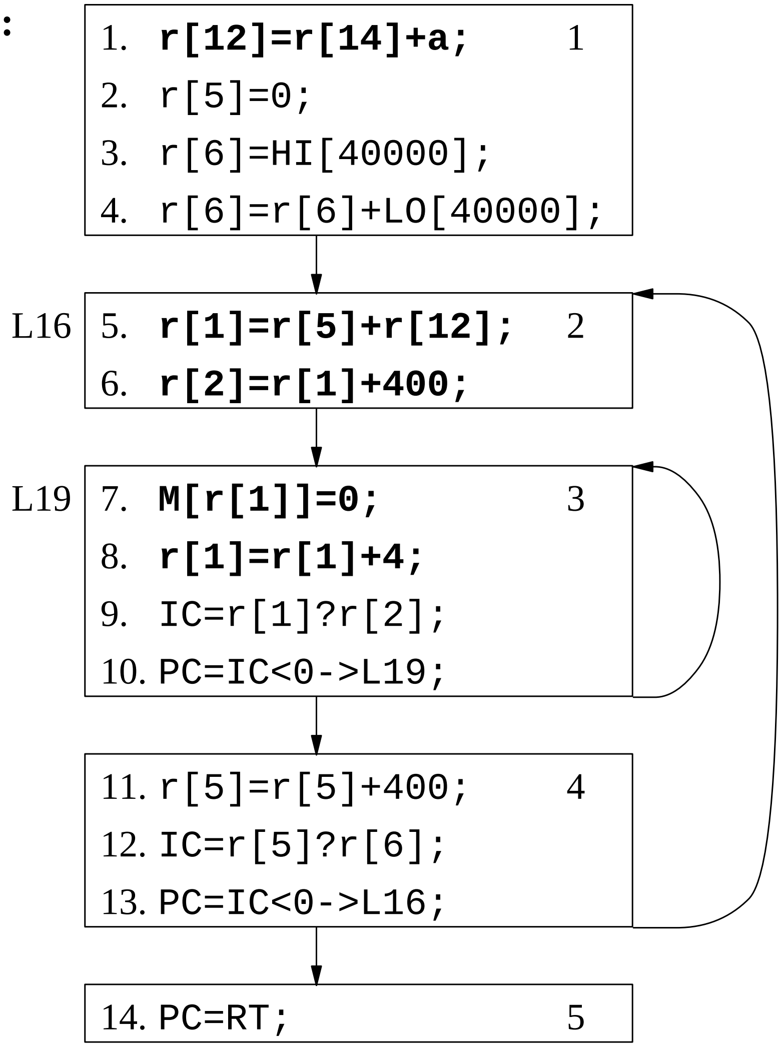
live range of d[] = [1,2,3,4,5,6,7,8] ∩ [5,6,7,8,9] = [5,6,7,8]

Determining Where Indirectly Taken Addresses are Dereferenced

Source Code:

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

Machine Instructions:



```
    1.  r[12]=r[14]+a;         1
    2.  r[5]=0;
    3.  r[6]=HI[40000];
    4.  r[6]=r[6]+LO[40000];

L16 5.  r[1]=r[5]+r[12];       2
    6.  r[2]=r[1]+400;

L19 7.  M[r[1]]=0;             3
    8.  r[1]=r[1]+4;
    9.  IC=r[1]?r[2];
    10. PC=IC<0->L19;

    11. r[5]=r[5]+400;         4
    12. IC=r[5]?r[6];
    13. PC=IC<0->L16;

    14. PC=RT;                 5
```

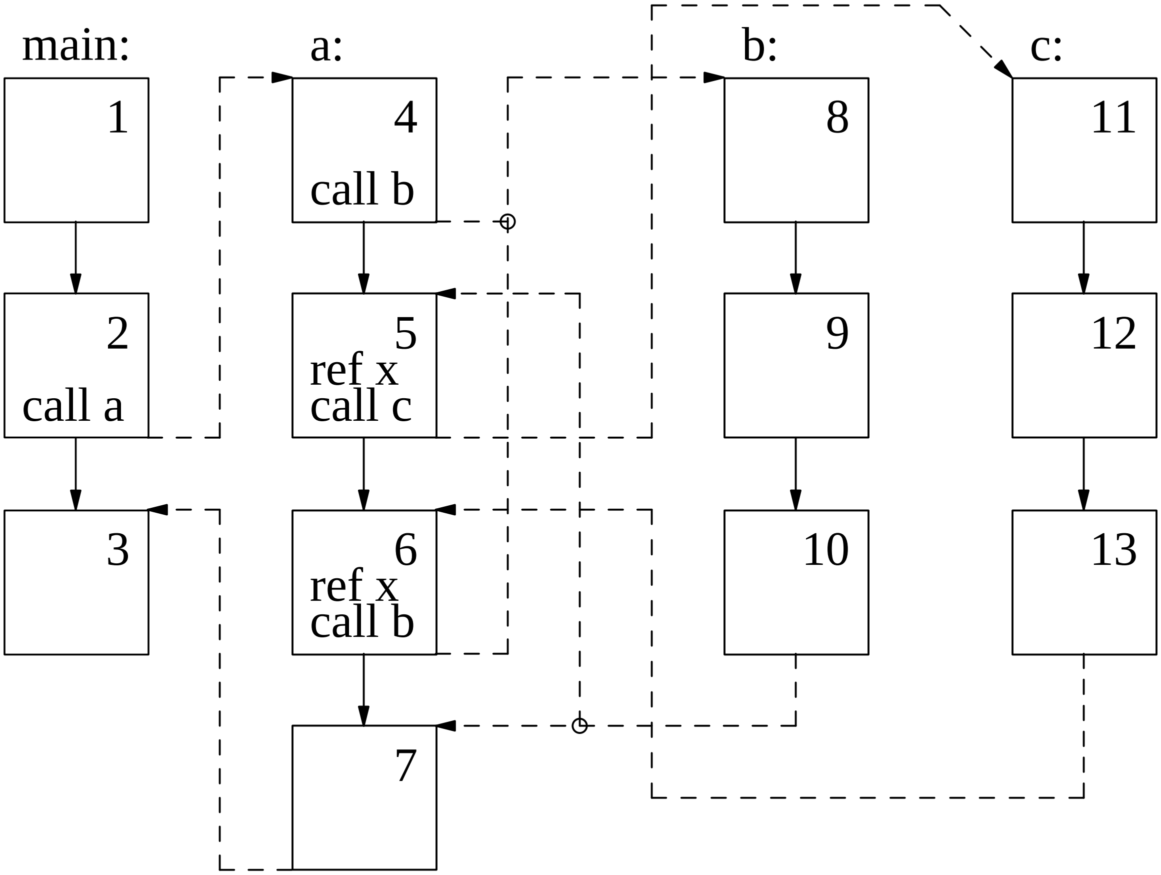
Detecting Live Ranges across Functions

- Calculate live ranges without propagating information into called functions.

initial live range of x = [1,2,4,5,6] $\cap$ [3,5,6,7] = [5,6]

- Include blocks within the functions that are called within the live range.

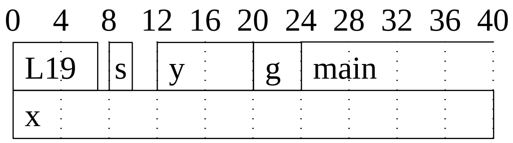
updated live range of x = [5,6] $\cup$ [11,12,13] = [5,6,11,12,13]



Assigning Variables to Memory Locations

```
int x[10];
int y[] = { 0, 1 };
int g = -1;
short s;
...
printf("Data: ");
```

(a) C Code Segment



(b) Offset Assignment

```
           .seg    "data"          !  switch to the data segment
           .global _x              !  make _x known to the linker
       _x:                         !  assoc _x address at offset 0
       L19:                        !  label of string at offset 0
           .ascii "Data: \0"       !  string value
           .skip   1               !  skip forward to offset 8 to align _s
           .global _s              !  make _s known to the linker
       _s:                         !  assoc _s address at offset 8
           .skip   2               !  skip forward to offset 12
           .global _y              !  make _y known to the linker
       _y:                         !  assoc _y address at offset 12
           .word   0               !  _y[0] set to 0
           .word   1               !  _y[1] set to 1
           .global _g              !  make _g known to the linker
       _g:                         !  assoc _g address at offset 20
           .word   -1              !  _g set to -1
           .global _main           !  make _main known to the linker
       _main:                      !  assoc _main address at offset 24
           save    %sp,-96,%sp  !  first inst within _main
           ...                     !  rest of insts in relocatable portion
           .seg    "text"          !  switch to the code segment
           ...                     !  all insts not overlapped with data
```

(c) SPARC Assembly Directives and Code

Overlapping Instructions by Cross Jumping

- Performed on jumps and calls.
- The compiler examines sets and uses to allow cross jumping of noncontiguous sequences of instructions.

Before Cross Jumping

Call 1

```
...
r[9]=HI[L166];
r[10]=HI[_lineno];
r[8]=r[9]+LO[L166];
r[9]=M[r[10]+LO[_lineno]];
r[10]=1;
CALL _pfnote();
```

Call 2

```
...
CALL _pfnote();
```

Call 3

```
...
r[9]=HI[L318];
r[10]=HI[_lineno];
r[8]=r[9]+LO[L318];
r[9]=M[r[10]+LO[_lineno]];
r[10]=1;
CALL _pfnote();
```

•••

function entry

```
_pfnote:
r[14]=SV[r[14]-1120];
...
```

After Cross Jumping

Call 1

```
...
r[9]=HI[L166];
r[8]=r[9]+LO[L166];
CALL _newlabel();
```

Call 2

```
...
CALL _pfnote();
```

Call 3

```
...
r[9]=HI[L318];
r[8]=r[9]+LO[L318];
CALL _newlabel();
```

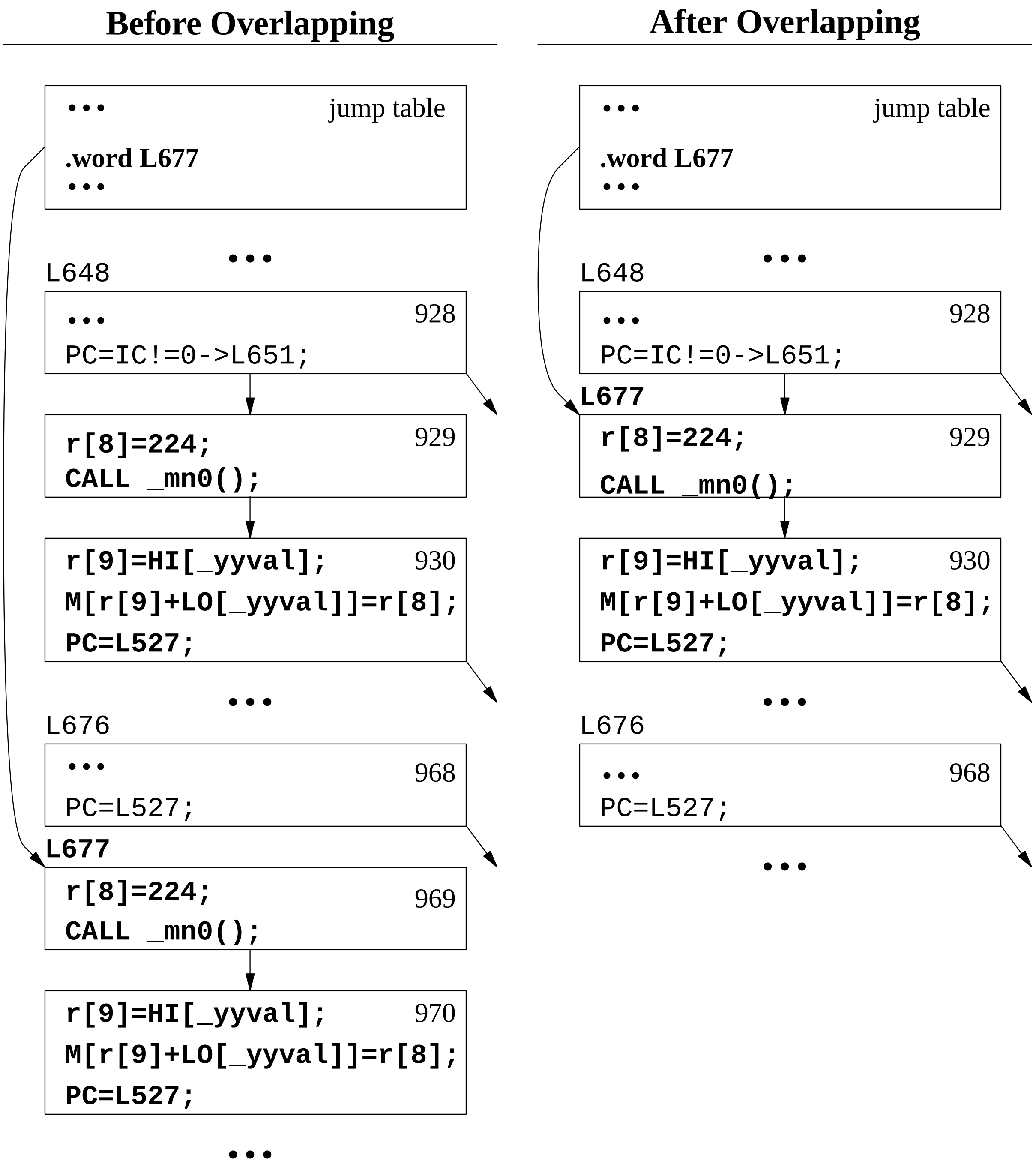
•••

function entry

```
_newlabel:
r[10]=HI[_lineno];
r[9]=M[r[10]+LO[_lineno]];
r[10]=1;
_pfnote:
r[14]=SV[r[14]-1120];
...
```

Abstracting Relocatable Code Portions

- Can overlap a relocatable code portion with a subset of another.



Before Overlapping

```
•••                          jump table
.word L677
•••

            •••
L648
•••                                 928
PC=IC!=0->L651;

r[8]=224;                           929
CALL _mn0();

r[9]=HI[_yyval];                    930
M[r[9]+LO[_yyval]]=r[8];
PC=L527;

            •••
L676
•••                                 968
PC=L527;
L677
r[8]=224;                           969
CALL _mn0();

r[9]=HI[_yyval];                    970
M[r[9]+LO[_yyval]]=r[8];
PC=L527;

            •••
```

After Overlapping

```
•••                          jump table
.word L677
•••

            •••
L648
•••                                 928
PC=IC!=0->L651;
L677
r[8]=224;                           929
CALL _mn0();

r[9]=HI[_yyval];                    930
M[r[9]+LO[_yyval]]=r[8];
PC=L527;

            •••
L676
•••                                 968
PC=L527;

            •••
```

Overlapping Static Data and Instructions

- Nonconflicting relocatable code portions and uninitialized static data can be overlapped in the initialized data segment.

```
...
char string[432];

main(argc, argv)
char *argv[];
{
   int y, i, j;
   int m;

   if(argc < 2) {
      printf(...);
      exit(0);
   }
...
   m = number(argv[1]);
...
   cal(m,y,string,24);
...
}

number(str)
char *str;
{
...
}
...
```

(a) Portion of *cal* Program

name	address range	num bytes	bytes saved
_string	000-431	432	0
L31	000-024	25	25
L74	025-038	14	14
L43	039-048	10	10
L55	049-056	8	8
L54	057-060	4	4
L44	061-064	4	4
L56	065-066	2	2
block range	address range	num bytes	bytes saved
1-3	068-103	36	36
42-44	104-123	20	20
45-45	124-135	12	12
46-50	136-199	64	64
51-51	200-207	8	8
4-18	268-483	216	164

(b) Mapping **`string`** with Static Data and Relocatable Code Segments

Results after Inlining and Cloning

- Code increasing transformations provide additional overlapping opportunities.

Program	Overlapping Run-Time Stack Data with Inlining		Overlapping Instructions with Cloning	
	Bytes Orig	Pct Less	Bytes Orig	Pct Less
cal	232	3.45%	1868	18.42%
cmp	192	0.00%	1576	-0.25%
csplit	728	0.00%	7988	1.85%
ctags	24544	0.36%	10308	0.50%
dhrystone	200	4.00%	2000	2.00%
grep	304	0.00%	4604	1.65%
join	96	0.00%	4280	0.93%
lex	7208	0.11%	44900	3.79%
linpack	3312	3.38%	11464	1.92%
mincost	192	4.17%	4500	3.64%
sdiff	5784	0.28%	7972	3.66%
tr	96	0.00%	1692	1.18%
tsp	2216	2.53%	4788	0.59%
whetstone	488	60.66%	4812	3.82%
yacc	1360	0.59%	32800	1.91%
average	3130	5.30%	9703	3.04%

Future Work

- Obtain more accurate live ranges of arrays.
- Overlap fields within a structure.
- Measure effect on unified secondary caches and paging.

Conclusions

- Overlapping uninitialized static data with static data and instructions was shown to be quite beneficial.
- Over 10% of the memory requirements of a program was eliminated.
- Code increasing transformations provide additional overlapping opportunities for instructions and run-time stack data.
- More accurate live range analysis of arrays should result in improved results.