Synchronous C + WCRT Algebra 101

Reinhard von Hanxleden
Joint work with Michael Mendler, Claus Traulsen, . . .

Real-Time and Embedded Systems Group (RTSYS)
Department of Computer Science
Christian-Albrechts-Universität zu Kiel
www.informatik.uni-kiel.de/rtsys

SYNCHRON 2011, Le Bois du Lys

Institut
für Informatik
Christian-Albrechts-Universität zu Kiel

Overview

The SC-Story From Last Time

How to get deterministic concurrency?

The SC-Story From Last Time

How to get deterministic concurrency?

- Deterministic decision on when to perform context switch
- Deterministic decision on what thread to switch to

The SC-Story From Last Time

How to get deterministic concurrency?

- Deterministic decision on when to perform context switch
- Deterministic decision on what thread to switch to

Idea: Cooperative thread scheduling at application level

The SC-Story From Last Time

How to get deterministic concurrency?

- Deterministic decision on when to perform context switch
- Deterministic decision on what thread to switch to

Idea: Cooperative thread scheduling at application level

- As in co-routines, threads decide *when* to resume another thread (static)
- However, a dispatcher decides *what* thread to resume (dynamic)

The SC-Story From Last Time

How to get deterministic concurrency?

- Deterministic decision on when to perform context switch
- Deterministic decision on what thread to switch to

Idea: Cooperative thread scheduling at application level

- As in co-routines, threads decide *when* to resume another thread (static)
- However, a dispatcher decides *what* thread to resume (dynamic)
- Reactive control flow implemented at application level—are still executing a sequential C program!

Producer

```
int main() {
  DEAD(28);
  volatile unsigned int * buf =
      (unsigned int*)(0x3F800200);
  unsigned int i = 0;
  for (i = 0; ; i++ ) {
    DEAD(26);
    *buf = i;
  }
  return 0;
}
```

Consumer

```
int main() {
  DEAD(41);
  volatile unsigned int * buf =
      (unsigned int*)(0x3F800200);
  unsigned int i = 0;
  int arr[8];
  for (i =0; i<8; i++)
    arr[i] = 0;
  for (i = 0; ; i++) {
    DEAD(26);
    register int tmp = *buf;
    arr[i%8] = tmp;
  }
  return 0;
}
```

Observer

```
int main() {
  DEAD(41);
  volatile unsigned int * buf =
      (unsigned int*)(0x3F800200);
  volatile unsigned int * fd =
      (unsigned int*)(0x80000600);
  unsigned int i = 0;
  for (i = 0; ; i++ ) {
    DEAD(26);
    *fd = *buf;
  }
  return 0;
}
```

Lickly *et al.*, *Predictable Programming on a Precision Timed Architecture*, CASES'08

Producer

```
int main() {
  DEAD(28);
  volatile unsigned int * buf =
      (unsigned int*)(0x3F800200);
  unsigned int i = 0;
  for (i = 0; ; i++ ) {
    DEAD(26);
    *buf = i;
  }
  return 0;
}
```

Consumer

```
int main() {
  DEAD(41);
  volatile unsigned int * buf =
      (unsigned int*)(0x3F800200);
  unsigned int i = 0;
  int arr[8];
  for (i =0; i<8; i++)
    arr[i] = 0;
  for (i = 0; ; i++) {
    DEAD(26);
    register int tmp = *buf;
    arr[i%8] = tmp;
  }
  return 0;
}
```

Observer

```
int main() {
  DEAD(41);
  volatile unsigned int * buf =
      (unsigned int*)(0x3F800200);
  volatile unsigned int * fd =
      (unsigned int*)(0x80000600);
  unsigned int i = 0;
  for (i = 0; ; i++ ) {
    DEAD(26);
    *fd = *buf;
  }
  return 0;
}
```

Lickly *et al.*, *Predictable Programming on a Precision Timed Architecture*, CASES'08

```
1  #include "sc.h"
2
3  // == MAIN FUNCTION ==
4  int main()
5  {
6    int notDone;
7
8    do {
9      notDone = tick();
10     //sleep(1);
11   } while (notDone);
12   return 0;
13 }
```

Producer

```
int main() {
  DEAD(28);
  volatile unsigned int * buf =
      (unsigned int*)(0x3F800200);
  unsigned int i = 0;
  for (i = 0; ; i++ ) {
    DEAD(26);
    *buf = i;
  }
  return 0;
}
```

Consumer

```
int main() {
  DEAD(41);
  volatile unsigned int * buf =
      (unsigned int*)(0x3F800200);
  unsigned int i = 0;
  int arr[8];
  for (i =0; i<8; i++)
    arr[i] = 0;
  for (i = 0; ; i++) {
    DEAD(26);
    register int tmp = *buf;
    arr[i%8] = tmp;
  }
  return 0;
}
```

Observer

```
int main() {
  DEAD(41);
  volatile unsigned int * buf =
      (unsigned int*)(0x3F800200);
  volatile unsigned int * fd =
      (unsigned int*)(0x80000600);
  unsigned int i = 0;
  for (i = 0; ; i++ ) {
    DEAD(26);
    *fd = *buf;
  }
  return 0;
}
```

Lickly *et al.*, *Predictable Programming on a Precision Timed Architecture*, CASES'08

```
1  #include "sc.h"
2
3  // == MAIN FUNCTION ==
4  int main()
5  {
6    int notDone;
7
8    do {
9      notDone = tick();
10     //sleep(1);
11   } while (notDone);
12   return 0;
13 }
```

```
14 // == TICK FUNCTION ==
15 int tick()
16 {
17   static int BUF, fd, i, j,
18     k = 0, tmp, arr[8];
19
20   TICKSTART(1);
21
22 PCO:
23   FORK(Producer, 3);
24   FORK(Consumer, 2);
25   FORKE(Observer);
26
27 Producer:
28   for (i = 0; ; i++) {
29     PAUSE;
30     BUF = i; }
```

```
31 Consumer:
32   for (j = 0; j < 8; j++)
33     arr[j] = 0;
34   for (j = 0; ; j++) {
35     PAUSE;
36     tmp = BUF;
37     arr[j % 8] = tmp; }
38
39 Observer:
40   for ( ; ; ) {
41     PAUSE;
42     fd = BUF;
43     k++; }
44
45   TICKEND;
46 }
```

Producer

```
int main() {
  DEAD(28);
  volatile unsigned int * buf =
      (unsigned int*)(0x3F800200);
  unsigned int i = 0;
  for (i = 0; ; i++ ) {
    DEAD(26);
    *buf = i;
  }
  return 0;
}
```

Consumer

```
int main() {
  DEAD(41);
  volatile unsigned int * buf =
      (unsigned int*)(0x3F800200);
  unsigned int i = 0;
  int arr[8];
  for (i =0; i<8; i++)
    arr[i] = 0;
  for (i = 0; ; i++) {
    DEAD(26);
    register int tmp = *buf;
    arr[i%8] = tmp;
  }
  return 0;
}
```

Observer

```
int main() {
  DEAD(41);
  volatile unsigned int * buf =
      (unsigned int*)(0x3F800200);
  volatile unsigned int * fd =
      (unsigned int*)(0x80000600);
  unsigned int i = 0;
  for (i = 0; ; i++ ) {
    DEAD(26);
    *fd = *buf;
  }
  return 0;
}
```

Lickly *et al.*, *Predictable Programming on a Precision Timed Architecture*, CASES'08

```
1  #include "sc.h"
2
3  // == MAIN FUNCTION ==
4  int main()
5  {
6    int notDone;
7
8    do {
9      notDone = tick();
10     //sleep(1);
11   } while (notDone);
12   return 0;
13 }
```

```
14 // == TICK FUNCTION ==
15 int tick()
16 {
17   static int BUF, fd, i, j,
18     k = 0, tmp, arr[8];
19
20   TICKSTART(1);
21
22   PCO:
23   FORK(Producer, 3);
24   FORK(Consumer, 2);
25   FORKE(Observer);
26
27   Producer:
28   for (i = 0; ; i++) {
29     PAUSE;
30     BUF = i; }
```

```
31   Consumer:
32   for (j = 0; j < 8; j++)
33     arr[j] = 0;
34   for (j = 0; ; j++) {
35     PAUSE;
36     tmp = BUF;
37     arr[j % 8] = tmp; }
38
39   Observer:
40   for ( ; ; ) {
41     PAUSE;
42     fd = BUF;
43     k++; }
44
45   TICKEND;
46 }
```

Producer

```
int main() {
  DEAD(28);
  volatile unsigned int * buf =
      (unsigned int*)(0x3F800200);
  unsigned int i = 0;
  for (i = 0; ; i++ ) {
    DEAD(26);
    *buf = i;
  }
  return 0;
}
```

Consumer

```
int main() {
  DEAD(41);
  volatile unsigned int * buf =
      (unsigned int*)(0x3F800200);
  unsigned int i = 0;
  int arr[8];
  for (i =0; i<8; i++)
    arr[i] = 0;
  for (i = 0; ; i++) {
    DEAD(26);
    register int tmp = *buf;
    arr[i%8] = tmp;
  }
  return 0;
}
```

Observer

```
int main() {
  DEAD(41);
  volatile unsigned int * buf =
      (unsigned int*)(0x3F800200);
  volatile unsigned int * fd =
      (unsigned int*)(0x80000600);
  unsigned int i = 0;
  for (i = 0; ; i++ ) {
    DEAD(26);
    *fd = *buf;
  }
  return 0;
}
```

Lickly *et al.*, *Predictable Programming on a Precision Timed Architecture*, CASES'08

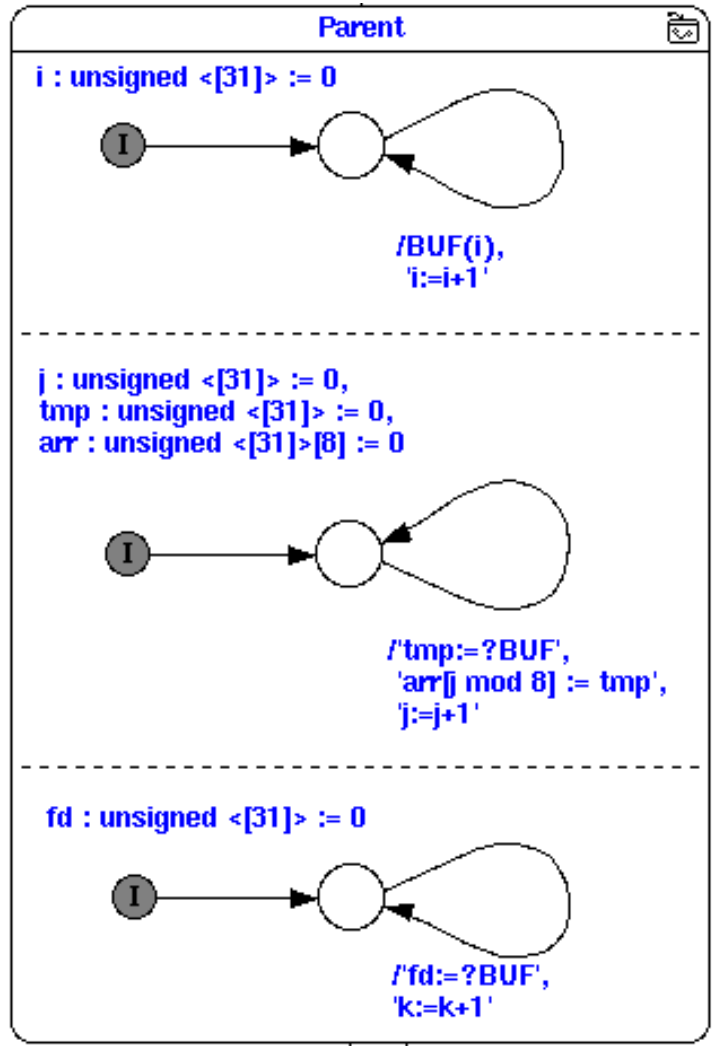
```
1  #include "sc.h"
2
3  // == MAIN FUNCTION ==
4  int main()
5  {
6    int notDone;
7
8    do {
9      notDone = tick();
10     //sleep(1);
11   } while (notDone);
12   return 0;
13 }
```

```
14 // == TICK FUNCTION ==
15 int tick()
16 {
17   static int BUF, fd, i, j,
18     k = 0, tmp, arr[8];
19
20   TICKSTART(1);
21
22  PCO:
23   FORK(Producer, 3);
24   FORK(Consumer, 2);
25   FORKE(Observer);
26
27  Producer:
28   for (i = 0; ; i++) {
29     PAUSE;
30     BUF = i; }
```

```
31  Consumer:
32   for (j = 0; j < 8; j++)
33     arr[j] = 0;
34   for (j = 0; ; j++) {
35     PAUSE;
36     tmp = BUF;
37     arr[j % 8] = tmp; }
38
39  Observer:
40   for ( ; ; ) {
41     PAUSE;
42     fd = BUF;
43     k++; }
44
45   TICKEND;
46 }
```

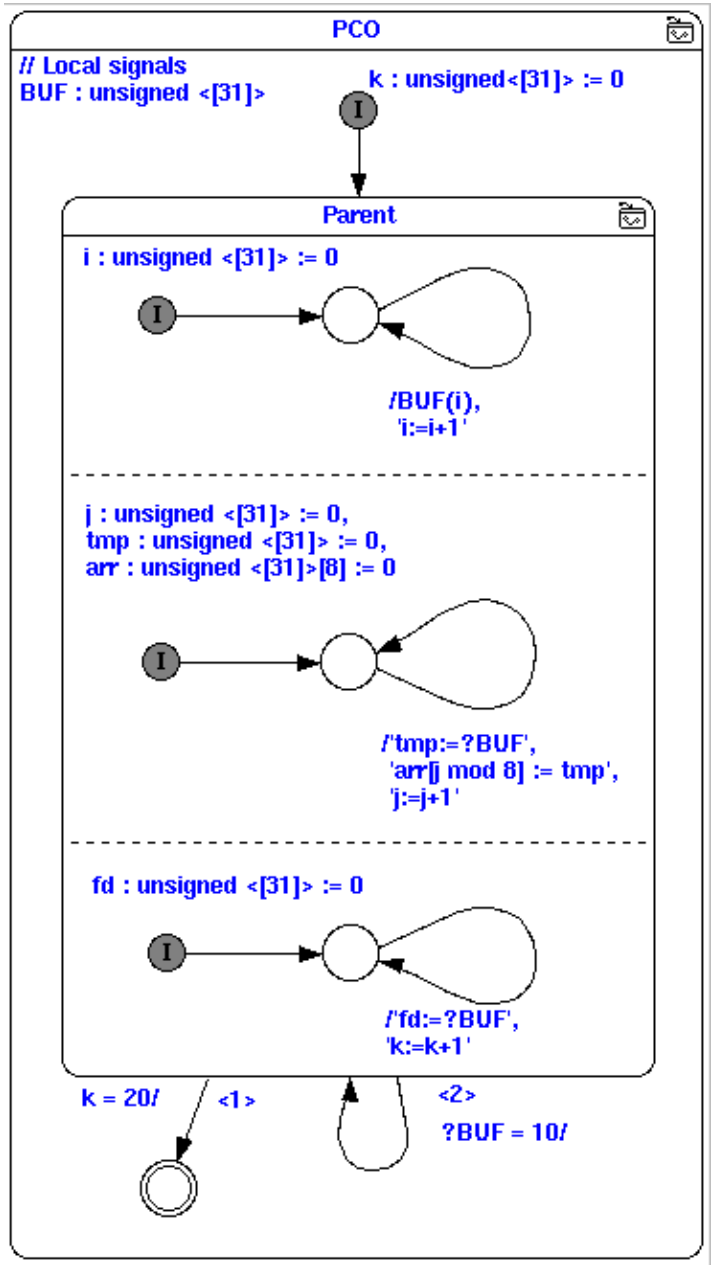
Producer-Consumer-Observer in SC

```
1   int tick()
2   {
3     static int BUF, fd, i, j,
4       k = 0, tmp, arr[8];
5
6     TICKSTART(1);
7
8     PCO:
9       FORK(Producer, 3);
10      FORK(Consumer, 2);
11      FORKE(Observer);
12
13    Producer:
14      for (i = 0; ; i++) {
15        PAUSE;
16        BUF = i; }
17
18    Consumer:
19      for (j = 0; j < 8; j++)
20        arr[j] = 0;
21      for (j = 0; ; j++) {
22        PAUSE;
23        tmp = BUF;
24        arr[j % 8] = tmp; }
25    Observer:
26      for ( ; ; ) {
27        PAUSE;
28        fd = BUF;
29        k++; }
30
31    TICKEND;
32  }
```



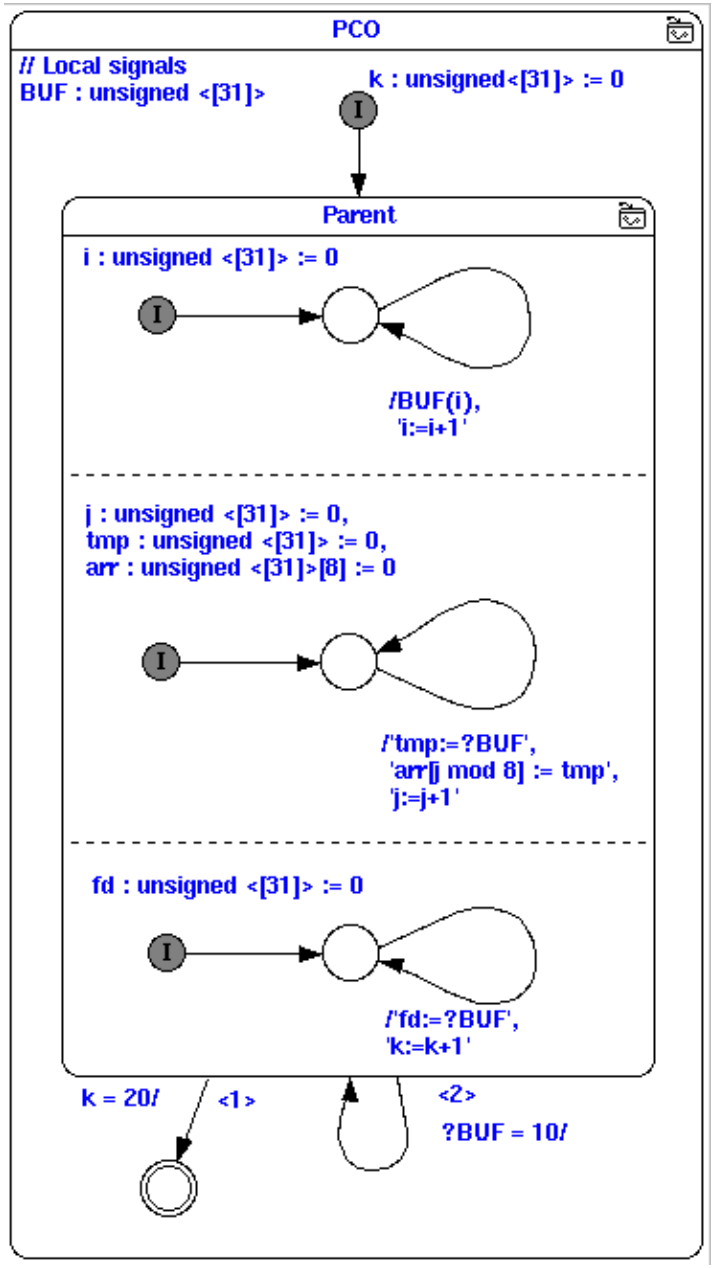
Producer-Consumer-Observer + Preemptions

```
1   int tick()
2   {
3     static int BUF, fd, i, j,
4       k = 0, tmp, arr[8];
5
6     TICKSTART(1);
7
8     PCO:
9       FORK(Producer, 3);
10      FORK(Consumer, 2);
11      FORKE(Observer);
12
13    Producer:
14      for (i = 0; ; i++) {
15        PAUSE;
16        BUF = i; }
17
18    Consumer:
19      for (j = 0; j < 8; j++)
20        arr[j] = 0;
21      for (j = 0; ; j++) {
22        PAUSE;
23        tmp = BUF;
24        arr[j % 8] = tmp; }
25    Observer:
26      for ( ; ; ) {
27        PAUSE;
28        fd = BUF;
29        k++; }
30
31    TICKEND;
32  }
```



Producer-Consumer-Observer + Preemptions in SC

```
1  int tick()
2  {
3    static int BUF, fd, i, j,
4      k = 0, tmp, arr[8];
5
6    TICKSTART(1);
7
8   PCO:
9    FORK(Producer, 4);
10   FORK(Consumer, 3);
11   FORK(Observer, 2);
12   FORKE(Parent);
13
14  Producer:
15   for (i = 0; ; i++) {
16     BUF = i;
17     PAUSE; }
18
19  Consumer:
20   for (j = 0; j < 8; j++)
21     arr[j] = 0;
22   for (j = 0; ; j++) {
23     tmp = BUF;
24     arr[j % 8] = tmp;
25     PAUSE; }
26  Observer:
27   for ( ; ; ) {
28     fd = BUF;
29     k++;
30     PAUSE; }
31
32  Parent:
33   while (1) {
34     if (k == 20)
35       TRANS(Done);
36     if (BUF == 10)
37       TRANS(PCO);
38     PAUSE;
39   }
40
41  Done:
42   TERM;
43   TICKEND;
44 }
```



Going on the Road . . .

Criticisms of Original SC

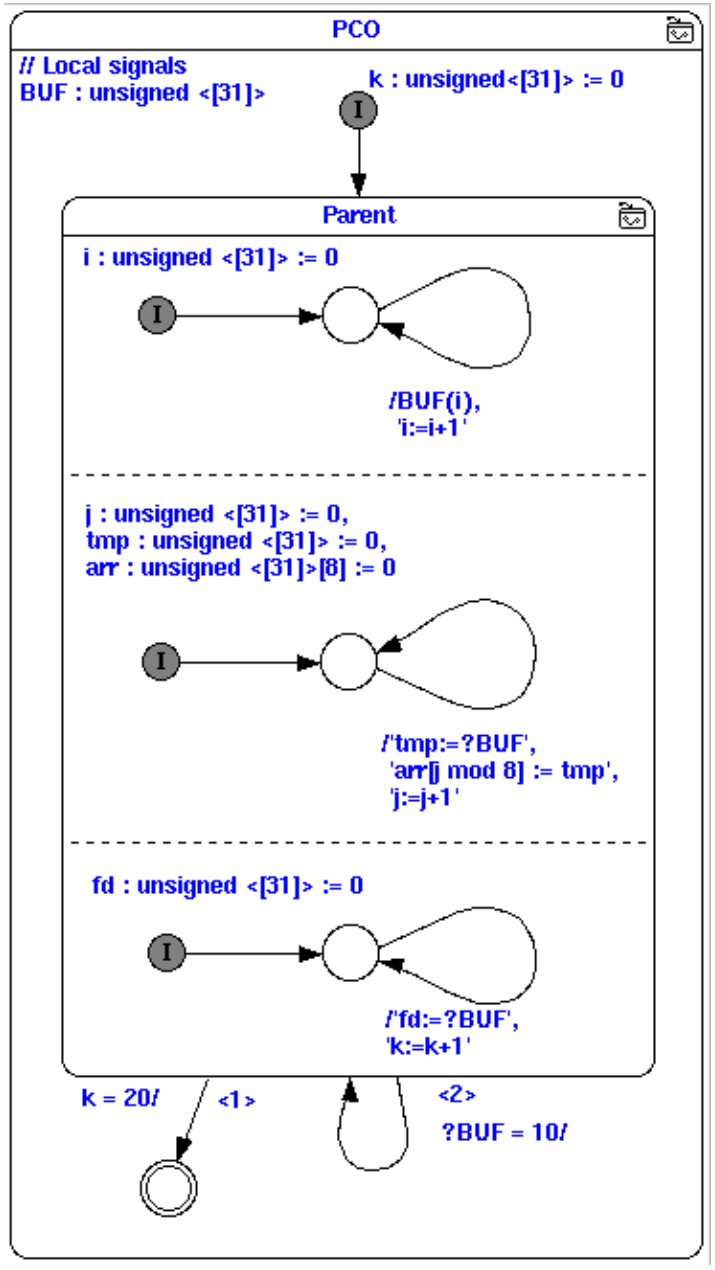
- Hard to understand what's going on
- Thread structure gets lost
- What does FORK/FORKE mean?

Proposed Alternative Syntax

- Replace labels by explicit "Thread" and "State" declarations
- Add syntactic scopes (braces)
- Predefined FORK1/FORK2/... operators

Original PCO-Example

```
1  int tick()
2  {
3    static int BUF, fd, i, j,
4      k = 0, tmp, arr[8];
5
6    TICKSTART(1);
7
8   PCO:
9    FORK(Producer, 4);
10   FORK(Consumer, 3);
11   FORK(Observer, 2);
12   FORKE(Parent);
13
14  Producer:
15   for (i = 0; ; i++) {
16     BUF = i;
17     PAUSE; }
18
19  Consumer:
20   for (j = 0; j < 8; j++)
21     arr[j] = 0;
22   for (j = 0; ; j++) {
23     tmp = BUF;
24     arr[j % 8] = tmp;
25     PAUSE; }
26  Observer:
27   for ( ; ; ) {
28     fd = BUF;
29     k++;
30     PAUSE; }
31
32  Parent:
33   while (1) {
34     if (k == 20)
35       TRANS(Done);
36     if (BUF == 10)
37       TRANS(PCO);
38     PAUSE;
39   }
40
41  Done:
42   TERM;
43   TICKEND;
44 }
```



Transforming the PCO-Example (1)

```
1  int tick()
2  {
3    static int BUF, fd, i, j,
4      k = 0, tmp, arr[8];
5
6    TICKSTART(1);
7
8   PCO:
9    FORK(Producer, 4);
10   FORK(Consumer, 3);
11   FORK(Observer, 2);
12   FORKE(Parent);
13
14  Parent:
15   while (1) {
16     if (k == 20)
17       TRANS(Done);
18     if (BUF == 10)
19       TRANS(PCO);
20     PAUSE;
21   }
22
23  Done:
24   TERM;
```

```
25  Producer:
26   for (i = 0; ; i++) {
27     BUF = i;
28     PAUSE; }
29
30  Consumer:
31   for (j = 0; j < 8; j++)
32     arr[j] = 0;
33   for (j = 0; ; j++) {
34     tmp = BUF;
35     arr[j % 8] = tmp;
36     PAUSE; }
37
38  Observer:
39   for ( ; ; ) {
40     fd = BUF;
41     k++;
42     PAUSE; }
43
44   TICKEND;
45 }
```

1. Separate threads

Transforming the PCO-Example (2)

```
1  int tick()
2  {
3    static int BUF, fd, i, j,
4      k = 0, tmp, arr[8];
5
6    TICKSTART(1);
7
8   PCO:
9    FORK3(Producer, 4,
10         Consumer, 3,
11         Observer, 2);
12
13   while (1) {
14     if (k == 20)
15       TRANS(Done);
16     if (BUF == 10)
17       TRANS(PCO);
18     PAUSE;
19   }
20
21  Done:
22   TERM;
```

```
23  Producer:
24   for (i = 0; ; i++) {
25     BUF = i;
26     PAUSE; }
27
28  Consumer:
29   for (j = 0; j < 8; j++)
30     arr[j] = 0;
31   for (j = 0; ; j++) {
32     tmp = BUF;
33     arr[j % 8] = tmp;
34     PAUSE; }
35
36  Observer:
37   for ( ; ; ) {
38     fd = BUF;
39     k++;
40     PAUSE; }
41
42   TICKEND;
43 }
```

1. Separate threads
2. Consolidate FORK

Transforming the PCO-Example (3)

```
1  int tick()
2  {
3    static int BUF, fd, i, j,
4      k = 0, tmp, arr[8];
5
6    MainThread (1) {
7    PCO:
8      FORK3(Producer, 4,
9            Consumer, 3,
10           Observer, 2);
11
12     while (1) {
13       if (k == 20)
14         TRANS(Done);
15       if (BUF == 10)
16         TRANS(PCO);
17       PAUSE;
18     }
19
20   Done:
21     TERM;
22   }
```

```
23   Thread (Producer) {
24     for (i = 0; ; i++) {
25       BUF = i;
26       PAUSE; }
27   }
28
29   Thread (Consumer) {
30     for (j = 0; j < 8; j++)
31       arr[j] = 0;
32     for (j = 0; ; j++) {
33       tmp = BUF;
34       arr[j % 8] = tmp;
35       PAUSE; }
36   }
37
38   Thread (Observer) {
39     for ( ; ; ) {
40       fd = BUF;
41       k++;
42       PAUSE; }
43   }
44
45   TICKEND;
46 }
```

1. Separate threads
2. Consolidate FORK
3. Add Thread scopes

Transforming the PCO-Example (4)

```
1  int tick()
2  {
3    static int BUF, fd, i, j,
4      k = 0, tmp, arr[8];
5
6    MainThread (1) {
7      State (PCO) {
8        FORK3(Producer, 4,
9              Consumer, 3,
10             Observer, 2);
11
12       while (1) {
13         if (k == 20)
14           TRANS(Done);
15         if (BUF == 10)
16           TRANS(PCO);
17         PAUSE;
18       }
19     }
20
21     State (Done) {
22       TERM;
23     }
24   }
```

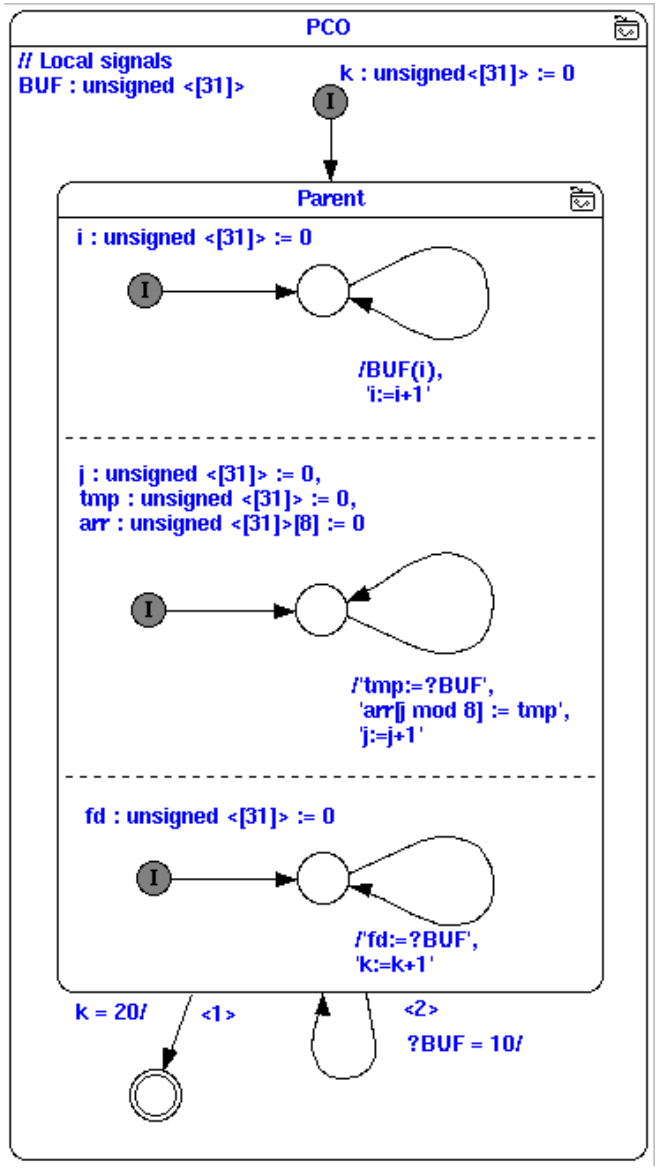
```
25   Thread (Producer) {
26     for (i = 0; ; i++) {
27       BUF = i;
28       PAUSE; }
29   }
30
31   Thread (Consumer) {
32     for (j = 0; j < 8; j++)
33       arr[j] = 0;
34     for (j = 0; ; j++) {
35       tmp = BUF;
36       arr[j % 8] = tmp;
37       PAUSE; }
38   }
39
40   Thread (Observer) {
41     for ( ; ; ) {
42       fd = BUF;
43       k++;
44       PAUSE; }
45   }
46
47   TICKEND;
48 }
```

1. Separate threads
2. Consolidate FORK
3. Add Thread scopes
4. Add States

Transformed PCO-Example

```
1  int tick()
2  {
3    static int BUF, fd, i, j,
4      k = 0, tmp, arr[8];
5
6    MainThread(1) {
7      State(PCO) {
8        FORK3(Producer, 4,
9              Consumer, 3,
10             Observer, 2);
11
12       while (1) {
13         if (k == 20)
14           TRANS(Done);
15         if (BUF == 10)
16           TRANS(PCO);
17         PAUSE;
18       }
19     }
20
21     State(Done) {
22       TERM;
23     }
24   }
```

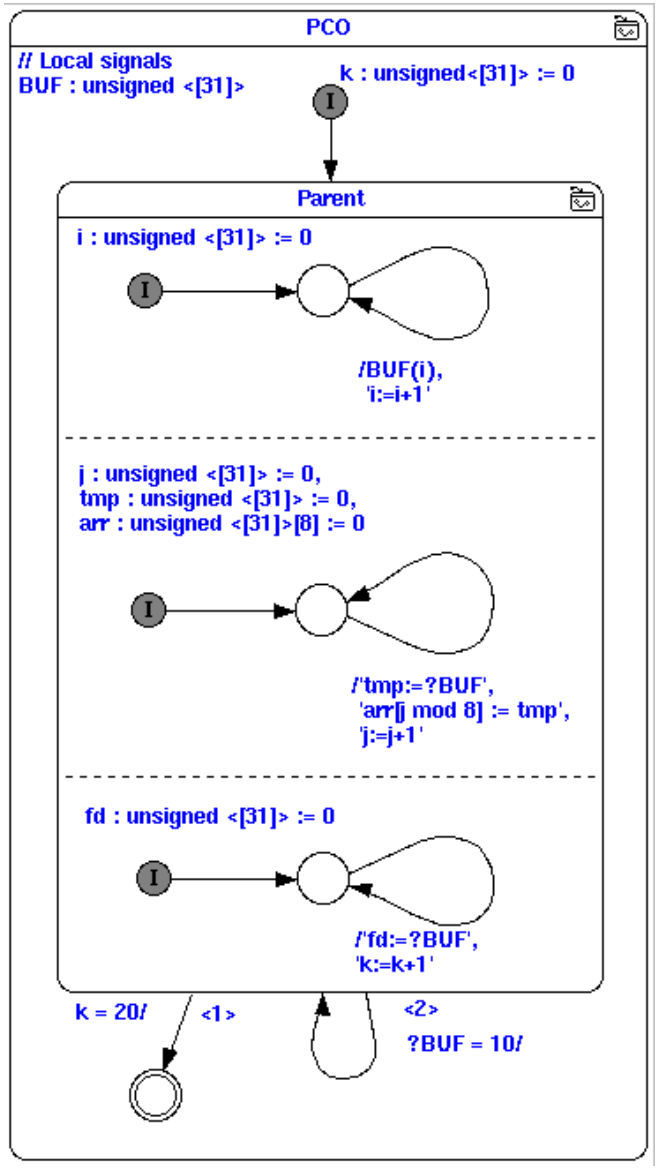
```
25   Thread(Producer) {
26     for (i = 0; ; i++) {
27       BUF = i;
28       PAUSE; }
29   }
30
31   Thread(Consumer) {
32     for (j = 0; j < 8; j++)
33       arr[j] = 0;
34     for (j = 0; ; j++) {
35       tmp = BUF;
36       arr[j % 8] = tmp;
37       PAUSE; }
38   }
39
40   Thread(Observer) {
41     for ( ; ; ) {
42       fd = BUF;
43       k++;
44       PAUSE; }
45   }
46
47   TICKEND;
48 }
```



Transformed PCO-Example

```
1  int tick()
2  {
3    static int BUF, fd, i, j,
4      k = 0, tmp, arr[8];
5
6    MainThread (1) {
7      State (PCO) {
8        FORK3(Producer, 4,
9              Consumer, 3,
10             Observer, 2);
11
12       while (1) {
13         if (k == 20)
14           TRANS(Done);
15         if (BUF == 10)
16           TRANS(PCO);
17         PAUSE;
18       }
19     }
20
21     State (Done) {
22       TERM;
23     }
24   }
```

```
25   Thread (Producer) {
26     for (i = 0; ; i++) {
27       BUF = i;
28       PAUSE; }
29   }
30
31   Thread (Consumer) {
32     for (j = 0; j < 8; j++)
33       arr[j] = 0;
34     for (j = 0; ; j++) {
35       tmp = BUF;
36       arr[j % 8] = tmp;
37       PAUSE; }
38   }
39
40   Thread (Observer) {
41     for ( ; ; ) {
42       fd = BUF;
43       k++;
44       PAUSE; }
45   }
46
47   TICKEND;
48 }
```



Still a sequential C program!

SC Summary

- Embeds reactive control flow constructs into C
  (Not discussed today: Synchronous Java)
- Light-weight + deterministic
- Multi-threaded, priority-based, co-routine like
- Full range of concurrency, preemption, signal handling

SC Summary

- Embeds reactive control flow constructs into C
  (Not discussed today: Synchronous Java)
- Light-weight + deterministic
- Multi-threaded, priority-based, co-routine like
- Full range of concurrency, preemption, signal handling

- Inspired by Kiel Esterel Processor (KEP)
  → Next part

References


Claus Traulsen, Torsten Amende, Reinhard von Hanxleden.
Compiling SyncCharts to Synchronous C.
DATE'11, Grenoble


Reinhard von Hanxleden.
SyncCharts in C—A Proposal for Light-Weight, Deterministic Concurrency.
EMSOFT'09, Grenoble

Outline

Worst Case Reaction Time (WCRT)

- Defined as upper bound for longest instantaneous path
- Measured e.g. in KEP instruction cycles
- Maximum time to react to given input with according output

Worst Case Reaction Time (WCRT)

- Defined as upper bound for longest instantaneous path
- Measured e.g. in KEP instruction cycles
- Maximum time to react to given input with according output

Usage of WCRT:

- Safely determine whether deadlines are met
- Can eliminate reaction time jitter of KEP by setting variable _TICKLEN according to WCRT

WCRT vs. WCET

Worst Case Execution Time

- Compute maximal execution time for piece of code

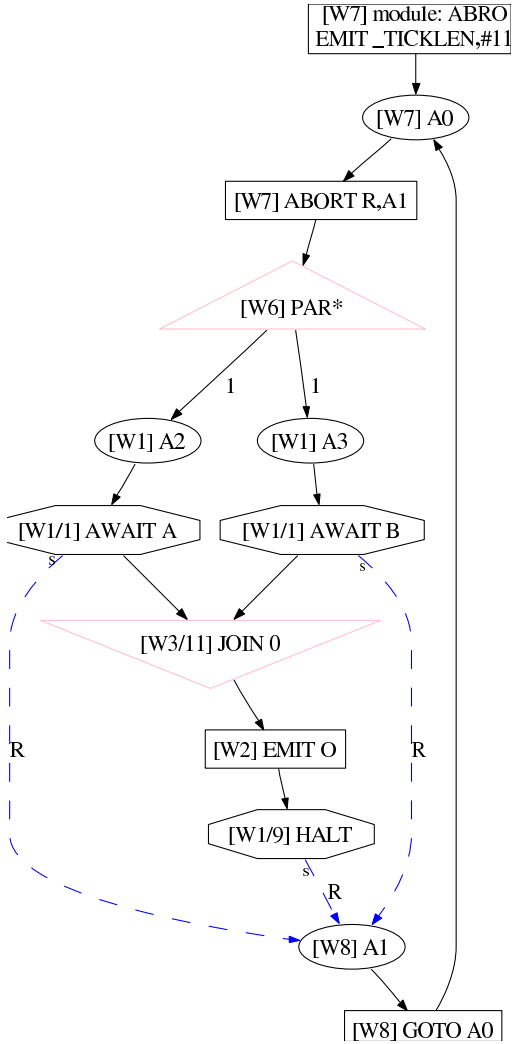
Worst Case Reaction Time

- Compute maximal time to react:
  one valid program state to another
- Similar to stabilization time of circuits

WCRT as Longest Path

```
     EMIT _TICKLEN,#11
A0:  ABORT R,A1
     PAR 1,A2,1
     PAR 1,A3,2
     PARE A4,1
A2:  AWAIT A
A3:  AWAIT B
A4:  JOIN 0
     EMIT O
     HALT
A1:  GOTO A0
```

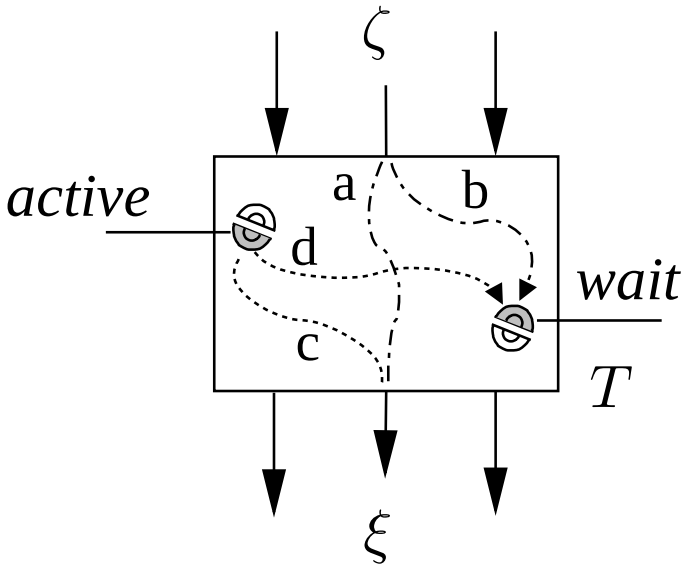
- Compute longest path between delay-nodes
- Abstract data-dependencies
- *Ad-hoc* optimizations



Interfaces

- Approach: use interface algebra to express WCRT
- Solid theoretical basis
- Allows refinement, eg. considering data-dependencies
- Modular computation (dynamic programming)
- Computation: $(max, +)$-algebra on timing matrix

Node Types



$$T = \begin{pmatrix} d_{thr} & d_{src} \\ d_{snk} & d_{int} \end{pmatrix} : (\zeta \vee active) \supset (\circ\xi \oplus \circ wait)$$

Interface Types

$$D{:}\phi \supset \psi$$

- Delay Matrix

$$D = \begin{pmatrix} d_{11} & \cdots & d_{1n} \\ \vdots & & \vdots \\ d_{m1} & \cdots & d_{mn} \end{pmatrix}$$

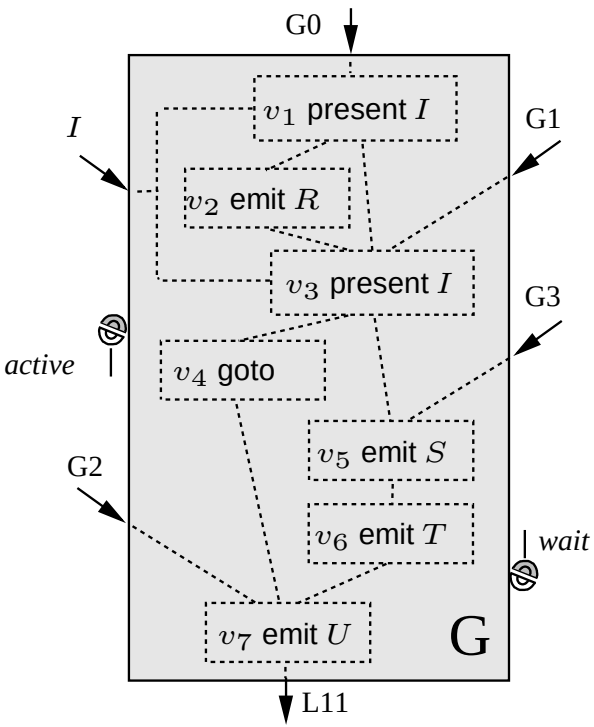
- Input Control

$$\phi = \zeta_1 \vee \zeta_2 \vee \cdots \vee \zeta_m$$

- Output Control

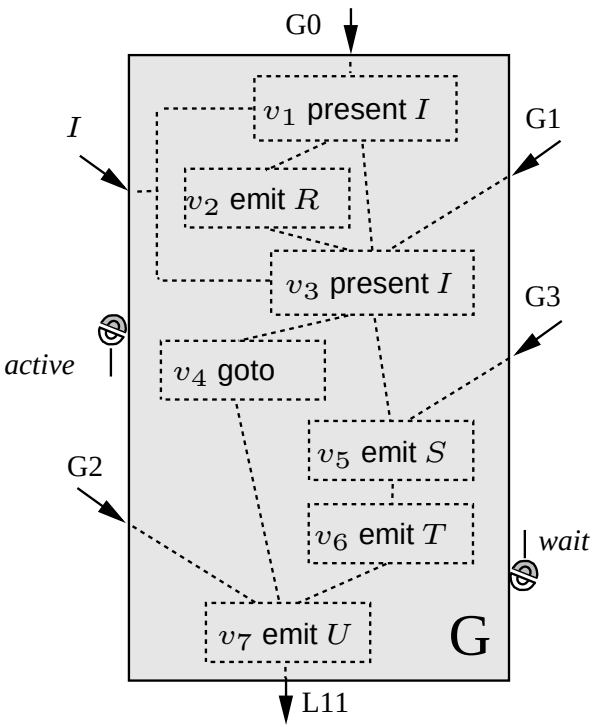
$$\psi = \circ\xi_1 \oplus \circ\xi_2 \oplus \cdots \oplus \circ\xi_n$$

Expressing the WCRT



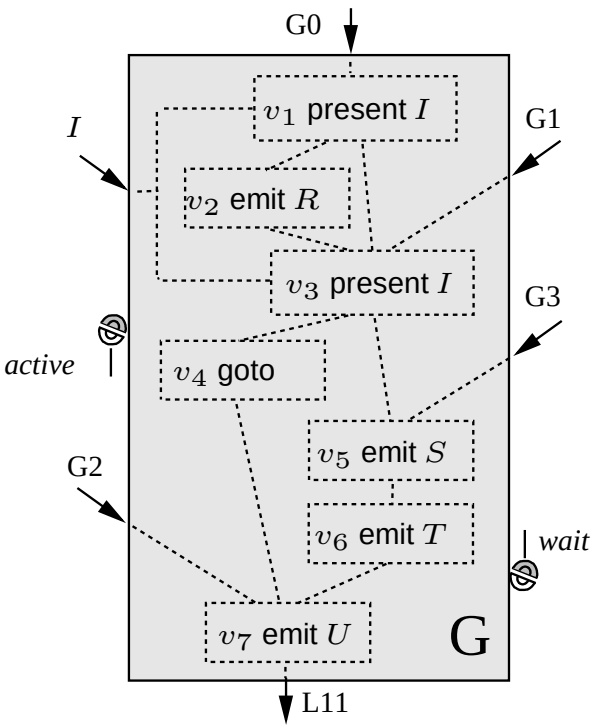
- $(6) : G0 \supset \circ L11$

Expressing the WCRT



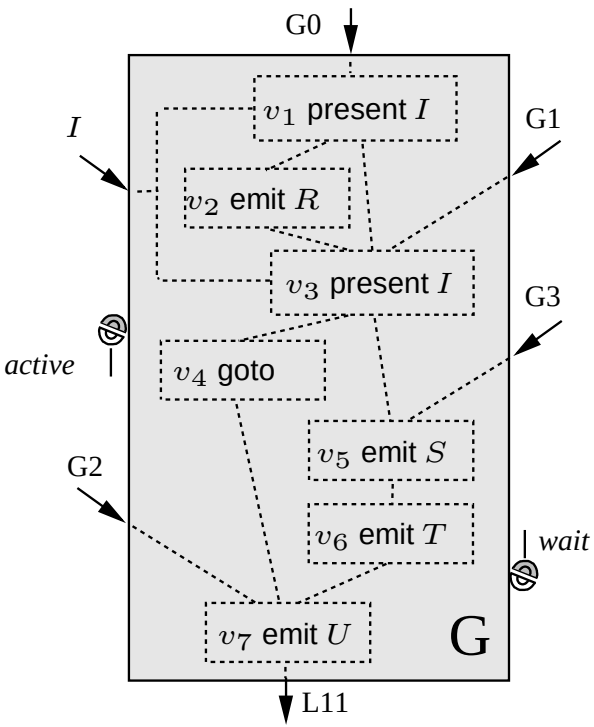
- $(6) : G0 \supset \circ L11$
- $(6, 4, 3, 1) :$
  $(G0 \vee G1 \vee G3 \vee G2) \supset \circ L11$

Expressing the WCRT



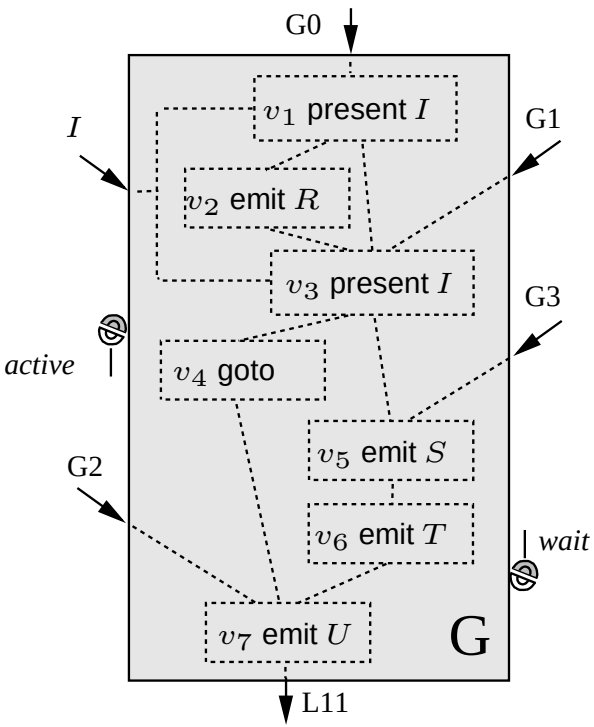
- $(6) : G0 \supset \circ L11$
- $(6,4,3,1) :$
  $(G0 \vee G1 \vee G3 \vee G2) \supset \circ L11$
- $(5,5,3,4,3,1) :$
  $((G0 \wedge I) \vee (G0 \wedge \neg I) \vee (G1 \wedge I) \vee$
  $(G1 \wedge \neg I) \vee G3 \vee G2) \supset \circ L11$

Expressing the WCRT



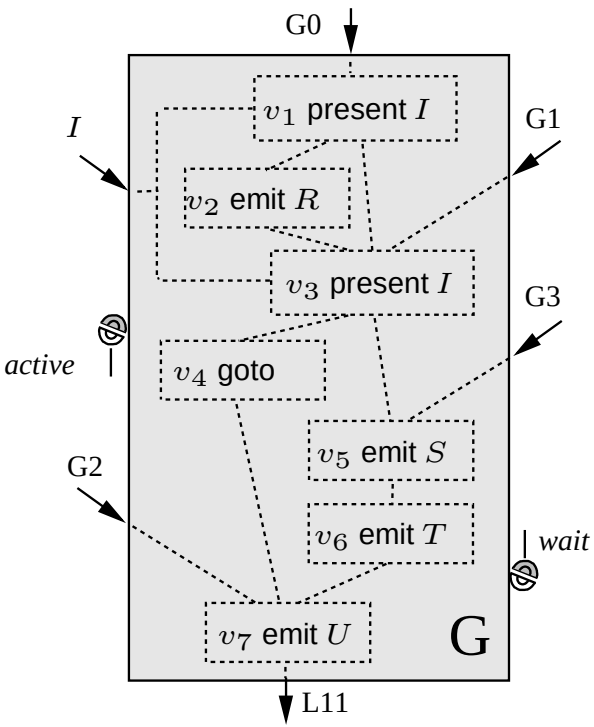
- $(6) : G0 \supset \circ L11$
- $(6, 4, 3, 1) :$
  $(G0 \vee G1 \vee G3 \vee G2) \supset \circ L11$
- $(5, 5, 3, 4, 3, 1) :$
  $((G0 \wedge I) \vee (G0 \wedge \neg I) \vee (G1 \wedge I) \vee$
  $(G1 \wedge \neg I) \vee G3 \vee G2) \supset \circ L11$
- $(5, 3, 4, 3, 1) : (G0 \vee (G1 \wedge I) \vee$
  $(G1 \wedge \neg I) \vee G3 \vee G2) \supset \circ L11$

Expressing the WCRT



- $(6) : G0 \supset \circ L11$
- $(6,4,3,1) :$
  $(G0 \vee G1 \vee G3 \vee G2) \supset \circ L11$
- $(5,5,3,4,3,1) :$
  $((G0 \wedge I) \vee (G0 \wedge \neg I) \vee (G1 \wedge I) \vee$
  $(G1 \wedge \neg I) \vee G3 \vee G2) \supset \circ L11$
- $(5,3,4,3,1) : (G0 \vee (G1 \wedge I) \vee$
  $(G1 \wedge \neg I) \vee G3 \vee G2) \supset \circ L11$
- $(5,3,4,1) : (G0 \vee ((G1 \wedge I) \oplus$
  $G3) \vee (G1 \wedge \neg I) \vee G2) \supset \circ L11$

Expressing the WCRT



- $(6) : G0 \supset \circ L11$
- $(6,4,3,1) :$
  $(G0 \vee G1 \vee G3 \vee G2) \supset \circ L11$
- $(5,5,3,4,3,1) :$
  $((G0 \wedge I) \vee (G0 \wedge \neg I) \vee (G1 \wedge I) \vee$
  $(G1 \wedge \neg I) \vee G3 \vee G2) \supset \circ L11$
- $(5,3,4,3,1) : (G0 \vee (G1 \wedge I) \vee$
  $(G1 \wedge \neg I) \vee G3 \vee G2) \supset \circ L11$
- $(5,3,4,1) : (G0 \vee ((G1 \wedge I) \oplus$
  $G3) \vee (G1 \wedge \neg I) \vee G2) \supset \circ L11$
- $(5) : G0 \supset \circ L11$

Summary/Outlook

- Algebraic approach allows to trade off precision and efficiency
- So far geared towards reactive processing ISA (KEP)
- Would like to lift this to higher level and mixed models (e.g. C/SC)

Summary/Outlook

- Algebraic approach allows to trade off precision and efficiency
- So far geared towards reactive processing ISA (KEP)
- Would like to lift this to higher level and mixed models (e.g. C/SC)

- To be continued . . . → talk by M. Mendler

References


Michael Mendler, Claus Traulsen, Reinhard von Hanxleden
WCRT Algebra and Interfaces for Esterel-Style Synchronous Processing
DATE'09, Nice


Partha S. Roop, Sidharta Andalam, Reinhard von Hanxleden, Simon Yuan, Claus Traulsen
Tight WCRT Analysis for Synchronous C Programs
CASES'09, Grenoble


Xin Li, Reinhard von Hanxleden
Multi-Threaded Reactive Programming—The Kiel Esterel Processor
IEEE Transactions on Computers 2011

Thanks!
Questions/Comments?

Appendix

Overview

Explaining the (Original) Title: SyncCharts ...

Reactive control flow:

- Sequentiality
- + Concurrency
- + Preemption

Explaining the (Original) Title: SyncCharts ...

Reactive control flow:

- Sequentiality
- + Concurrency
- + Preemption

Statecharts [Harel 1987]:

- Reactive control flow
- + Visual syntax

Explaining the (Original) Title: SyncCharts ...

Reactive control flow:

- Sequentiality
- + Concurrency
- + Preemption

Statecharts [Harel 1987]:

- Reactive control flow
- + Visual syntax

SyncCharts [André 1996]:

- Statecharts concept
- + Synchronous semantics

. . . in C

Today's Scenario 1: Develop model in SyncCharts, synthesize C

- ☺ Can use visual syntax

. . . in C

Today's Scenario 1: Develop model in SyncCharts, synthesize C

- ☺ Can use visual syntax
- ☹ Need special modeling tool
- ☹ Cannot directly use full power of classical imperative language

. . . in C

Today's Scenario 1: Develop model in SyncCharts, synthesize C

- ☺ Can use visual syntax
- ☹ Need special modeling tool
- ☹ Cannot directly use full power of classical imperative language

Today's Scenario 2: Program "State Machine Pattern" in C

- ☺ Just need regular compiler

. . . in C

Today's Scenario 1: Develop model in SyncCharts, synthesize C

- ☺ Can use visual syntax
- ☹ Need special modeling tool
- ☹ Cannot directly use full power of classical imperative language

Today's Scenario 2: Program "State Machine Pattern" in C

- ☺ Just need regular compiler
- ☹ Relies on scheduler of run time system—no determinism
- ☹ Typically rather heavyweight

. . . in C

Today's Scenario 1: Develop model in SyncCharts, synthesize C

- ☺ Can use visual syntax
- ☹ Need special modeling tool
- ☹ Cannot directly use full power of classical imperative language

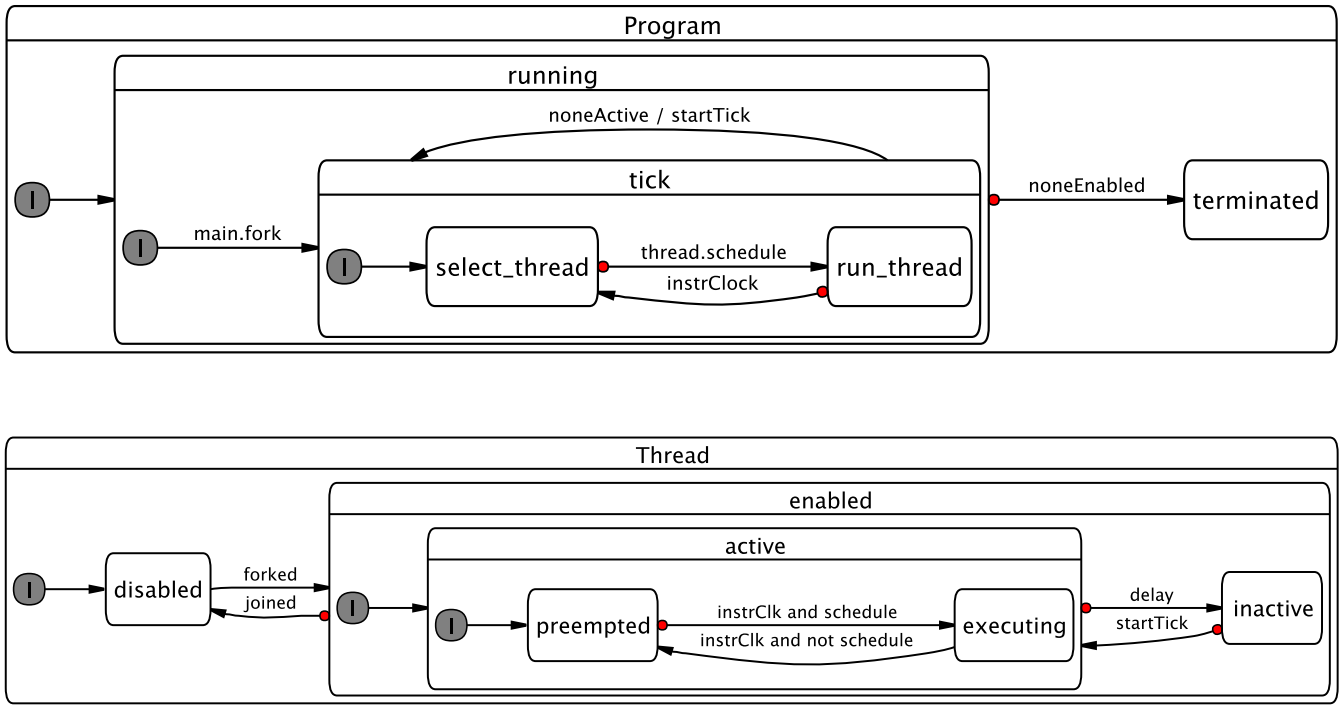
Today's Scenario 2: Program "State Machine Pattern" in C

- ☺ Just need regular compiler
- ☹ Relies on scheduler of run time system—no determinism
- ☹ Typically rather heavyweight

SyncCharts in C scenario: Use SC Operators

- ☺ Light weight to implement and to execute
- ☺ Just need regular compiler
- ☺ Semantics grounded in synchronous model

The inspiration: Reactive processing



Li *et al.*, ASPLOS'06

- SC multi-threading very close to Kiel Esterel Processor
- Difference: KEP dispatches at every instrClk, SC only at specific SC operators (such as PAUSE, PRIO)

SC Thread Operators

TICKSTART*(*init*, *p*)	Start (initial) tick, assign main thread priority *p*.
TICKEND	Return true (1) iff there is still an enabled thread.

SC Thread Operators

TICKSTART*(*init*, *p*)	Start (initial) tick, assign main thread priority *p*.
TICKEND	Return true (1) iff there is still an enabled thread.
PAUSE*+	Deactivate current thread for this tick.
TERM*	Terminate current thread.
ABORT	Abort descendant threads.
TRANS(*l*)	Shorthand for ABORT; GOTO(*l*).
SUSPEND*(*cond*)	Suspend (pause) thread + descendants if *cond* holds.

SC Thread Operators

TICKSTART*(*init*, *p*)	Start (initial) tick, assign main thread priority *p*.
TICKEND	Return true (1) iff there is still an enabled thread.
PAUSE*+	Deactivate current thread for this tick.
TERM*	Terminate current thread.
ABORT	Abort descendant threads.
TRANS(*l*)	Shorthand for ABORT; GOTO(*l*).
SUSPEND*(*cond*)	Suspend (pause) thread + descendants if *cond* holds.
FORK(*l*, *p*)	Create a thread with start address *l* and priority *p*.
FORKE*(*l*)	Finalize FORK, resume at *l*.
JOINELSE*+(l_{else})	If descendant threads have terminated normally, proceed; else pause, jump to l_{else}.

SC Thread Operators

TICKSTART*(*init*, *p*)	Start (initial) tick, assign main thread priority *p*.
TICKEND	Return true (1) iff there is still an enabled thread.
PAUSE*+	Deactivate current thread for this tick.
TERM*	Terminate current thread.
ABORT	Abort descendant threads.
TRANS(*l*)	Shorthand for ABORT; GOTO(*l*).
SUSPEND*(*cond*)	Suspend (pause) thread + descendants if *cond* holds.
FORK(*l*, *p*)	Create a thread with start address *l* and priority *p*.
FORKE*(*l*)	Finalize FORK, resume at *l*.
JOINELSE*+(l_{else})	If descendant threads have terminated normally, proceed; else pause, jump to l_{else}.

SC Signal Operators

SIGNAL(S)	Initialize a local signal S.
EMIT(S)	Emit signal S.
PRESENT(S, l_{else})	If S is present, proceed normally; else, jump to l_{else}.
EMITINT(S, *val*)	Emit valued signal S, of type integer, with value *val*.
EMITINTMUL(S, *val*)	Emit valued signal S, of type integer, combined with multiplication, with value *val*.
VAL(S, *reg*)	Retrieve value of signal S, into register/variable *reg*.
PRESENTPRE(S, l_{else})	If S was present in previous tick, proceed normally; else, jump to l_{else}. If S is a signal local to thread t, consider last preceeding tick in which t was active, *i. e.*, not suspended.
VALPRE(S, *reg*)	Retrieve value of signal S at previous tick, into register/variable *reg*.

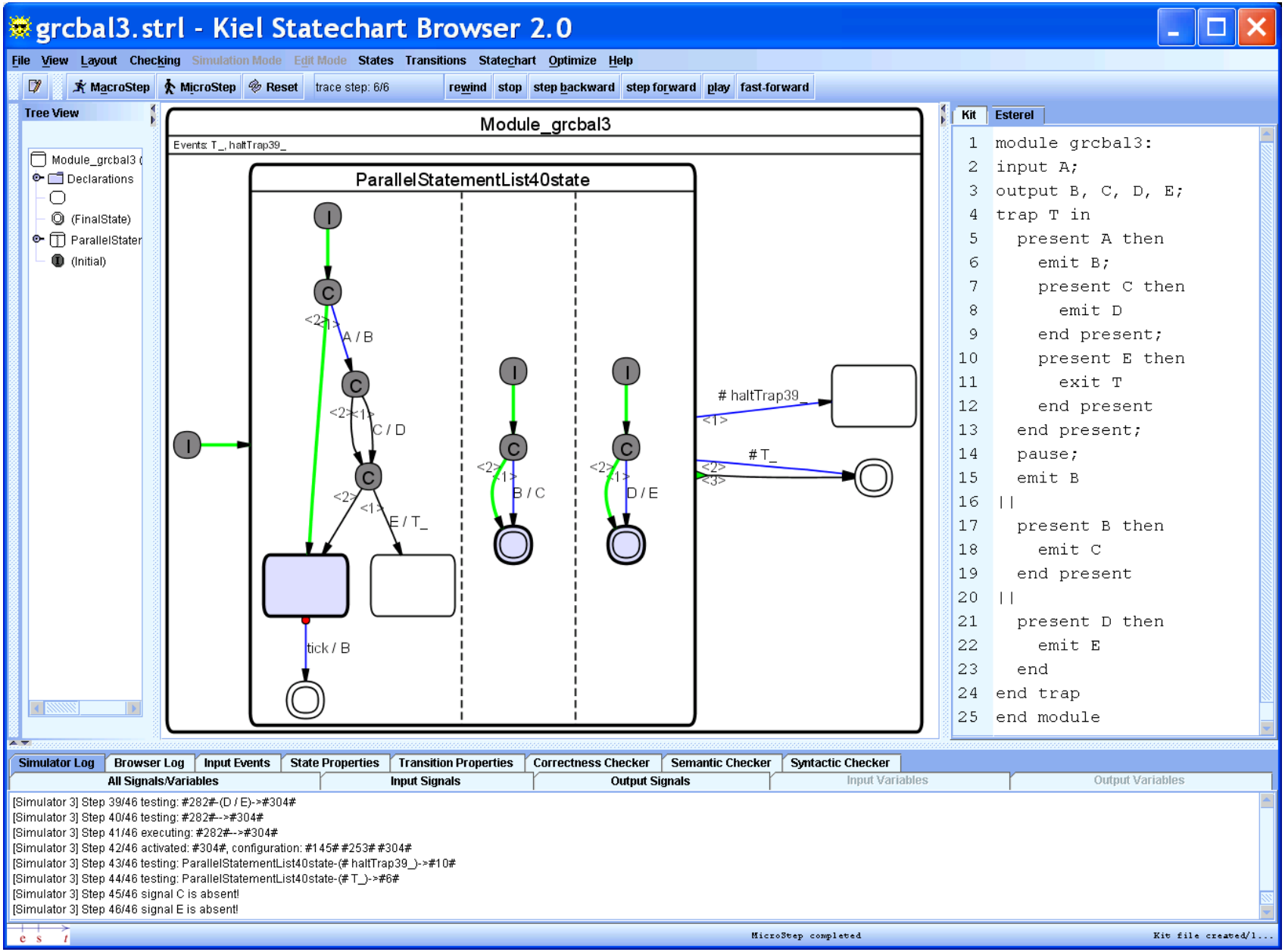
Further Operators

GOTO(l)	Jump to label l.
CALL(l, l_{ret})	Call function l (eg, an on exit function), return to l_{ret}.
RET	Return from function call.
ISAT(id, l_{state}, l)	If thread id is at state l_{state}, then proceed to next instruction (*e. g.*, an on exit function associated with id at state l_{state}). Else, jump to label l.
PPAUSE*(p, l)	Shorthand for PRIO(p, l'); l': PAUSE(l) (saves one call to dispatcher).
JPPAUSE*(p, l_{then}, l_{else})	Shorthand for JOIN(l_{then}, l); l: PPAUSE(p, l_{else}) (saves another call to dispatcher).
ISATCALL(id, l_{state}, l_{action}, l)	Shorthand for ISAT(id, l_{state}, l); CALL(l_{action}, l)

Thread Synchronization and Signals

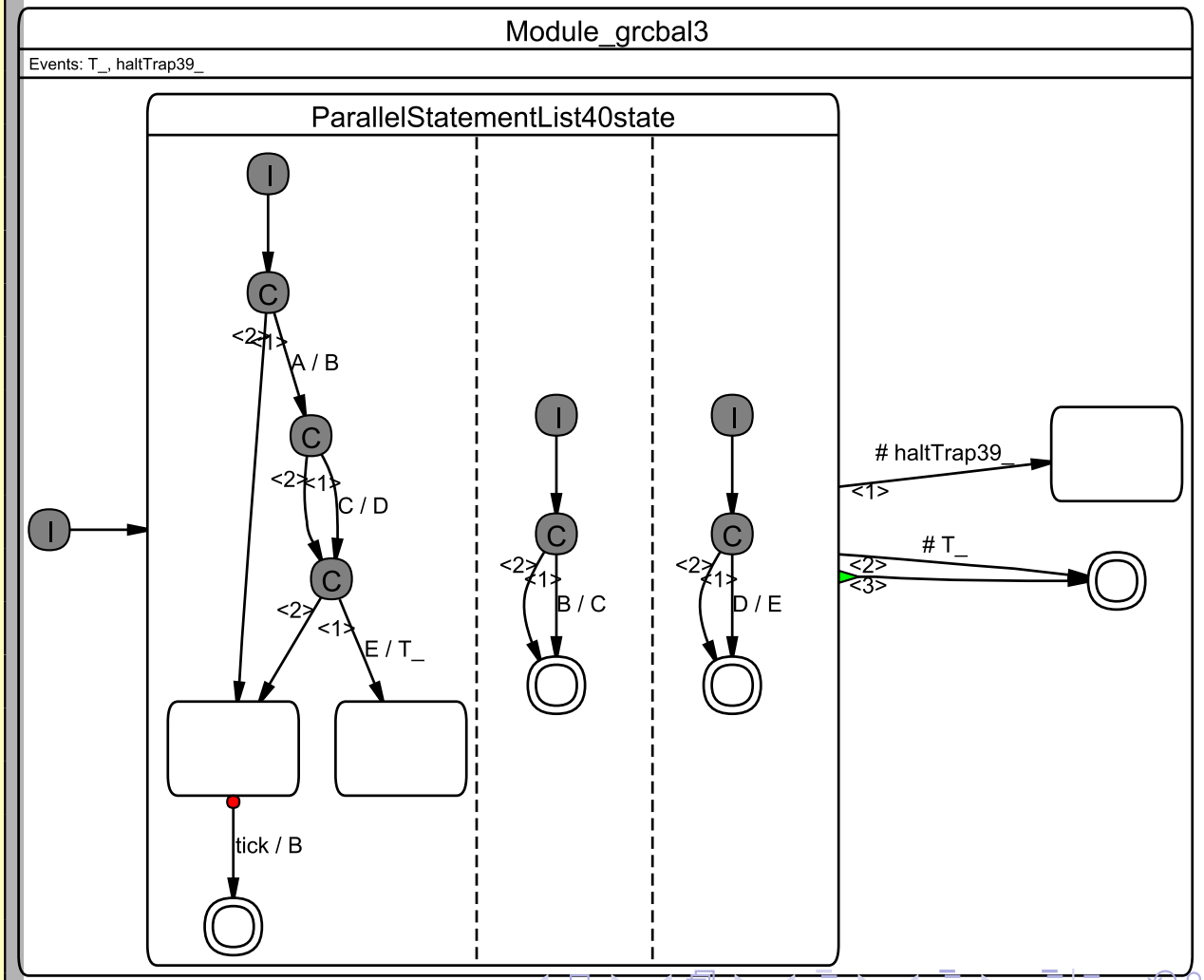
Recall: Threads may also communicate via signals

- In addition to thread operators, S provides signal operators (EMIT, PRESENT, PRE, valued/combined signals)
- Can handle signal dependencies and instantaneous communication via dynamic priorities

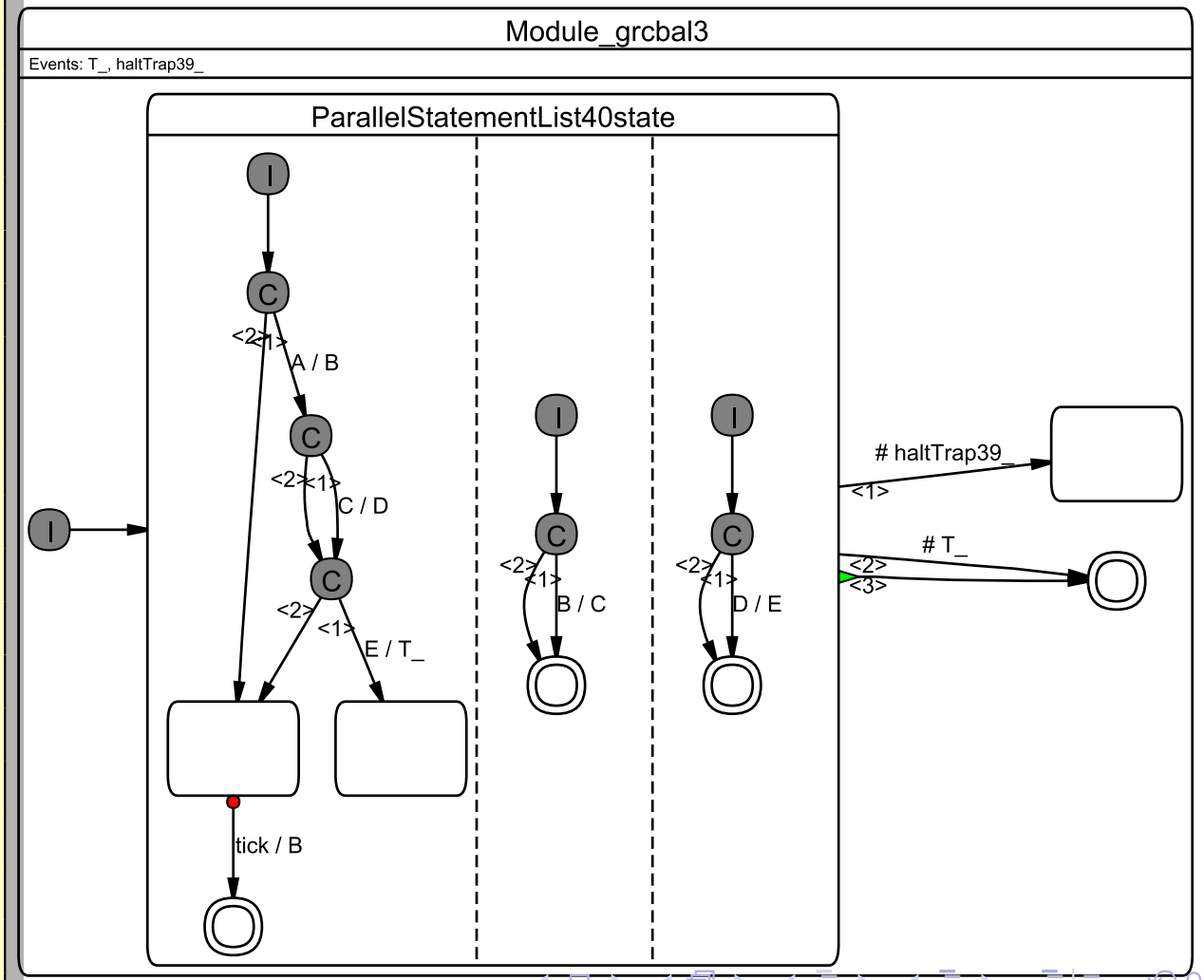


Edwards *et al.*, JES'07; Prochnow *et al.*, LCTES'06

```
 FORK(T1, 6);
 FORK(T2, 5);
 FORK(T3, 3);
 FORKE(TMain);

T1: if (PRESENT(A)) {
    EMIT(B);
    PRIO(4);
    if (PRESENT(C))
      EMIT(D);
    PRIO(2);
    if (PRESENT(E)) {
      EMIT(T_);
      TERM; }
 }
 PAUSE;
 EMIT(B);
 TERM;

T2: if (PRESENT(B))
    EMIT(C);
 TERM;

T3: if (PRESENT(D))
    EMIT(E);
 TERM;
```



```
 FORK(T1, 6);
 FORK(T2, 5);
 FORK(T3, 3);
 FORKE(TMain);

T1: if (PRESENT(A)) {
   EMIT(B);
   PRIO(4);
   if (PRESENT(C))
     EMIT(D);
   PRIO(2);
   if (PRESENT(E)) {
     EMIT(T_);
     TERM; }
 }
 PAUSE;
 EMIT(B);
 TERM;

T2: if (PRESENT(B))
   EMIT(C);
 TERM;

T3: if (PRESENT(D))
   EMIT(E);
 TERM;
```



```
 FORK(T1, 6);
 FORK(T2, 5);
 FORK(T3, 3);
 FORKE(TMain);

T1: if (PRESENT(A)) {
   EMIT(B);
   PRIO(4);
   if (PRESENT(C))
     EMIT(D);
   PRIO(2);
   if (PRESENT(E)) {
     EMIT(T_);
     TERM; }
 }
 PAUSE;
 EMIT(B);
 TERM;

T2: if (PRESENT(B))
   EMIT(C);
 TERM;

T3: if (PRESENT(D))
   EMIT(E);
 TERM;
```

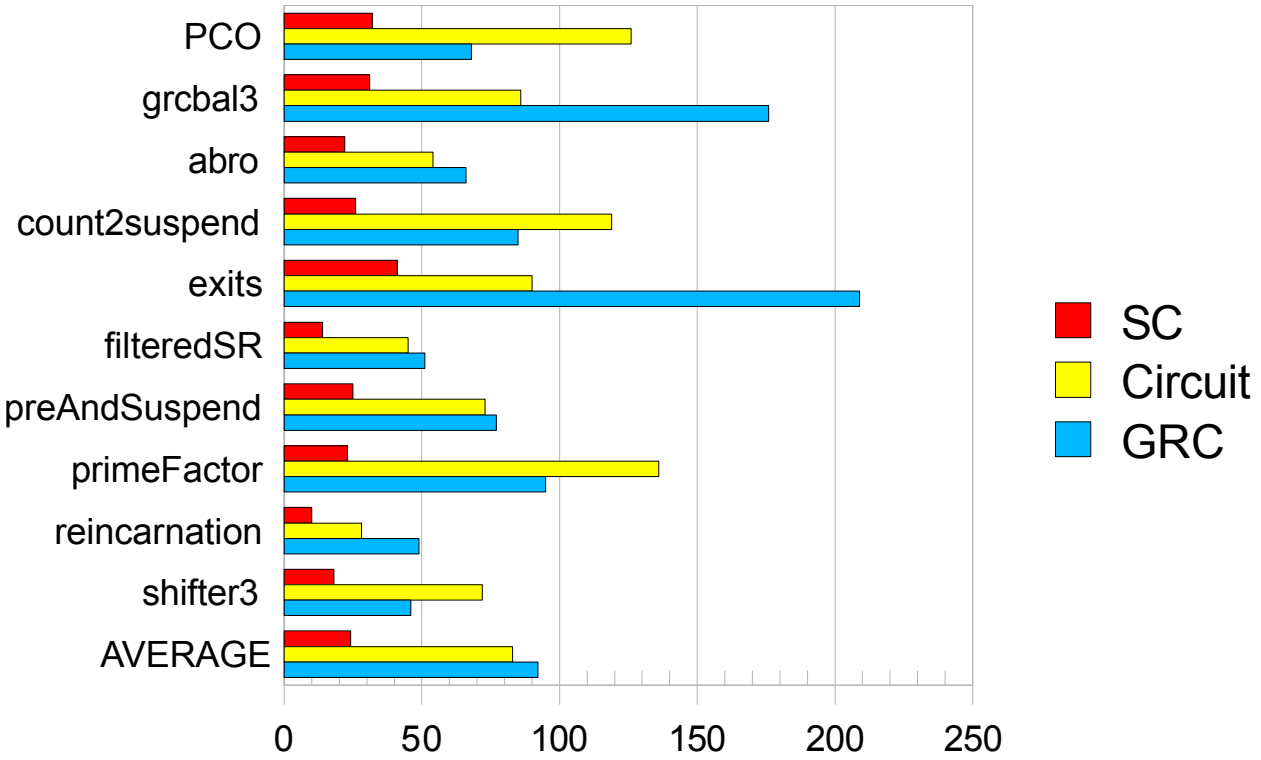
Sample Execution

```
FORK(T1, 6);
FORK(T2, 5);
FORK(T3, 3);
FORKE(TMain);

T1: if (PRESENT(A)) {
  EMIT(B);
  PRIO(4);
  if (PRESENT(C))
    EMIT(D);
  PRIO(2);
  if (PRESENT(E)) {
    EMIT(T_);
    TERM; }
}
PAUSE;
EMIT(B);
TERM;

T2: if (PRESENT(B))
  EMIT(C);
TERM;

T3: if (PRESENT(D))
  EMIT(E);
TERM;
```

Sample Execution

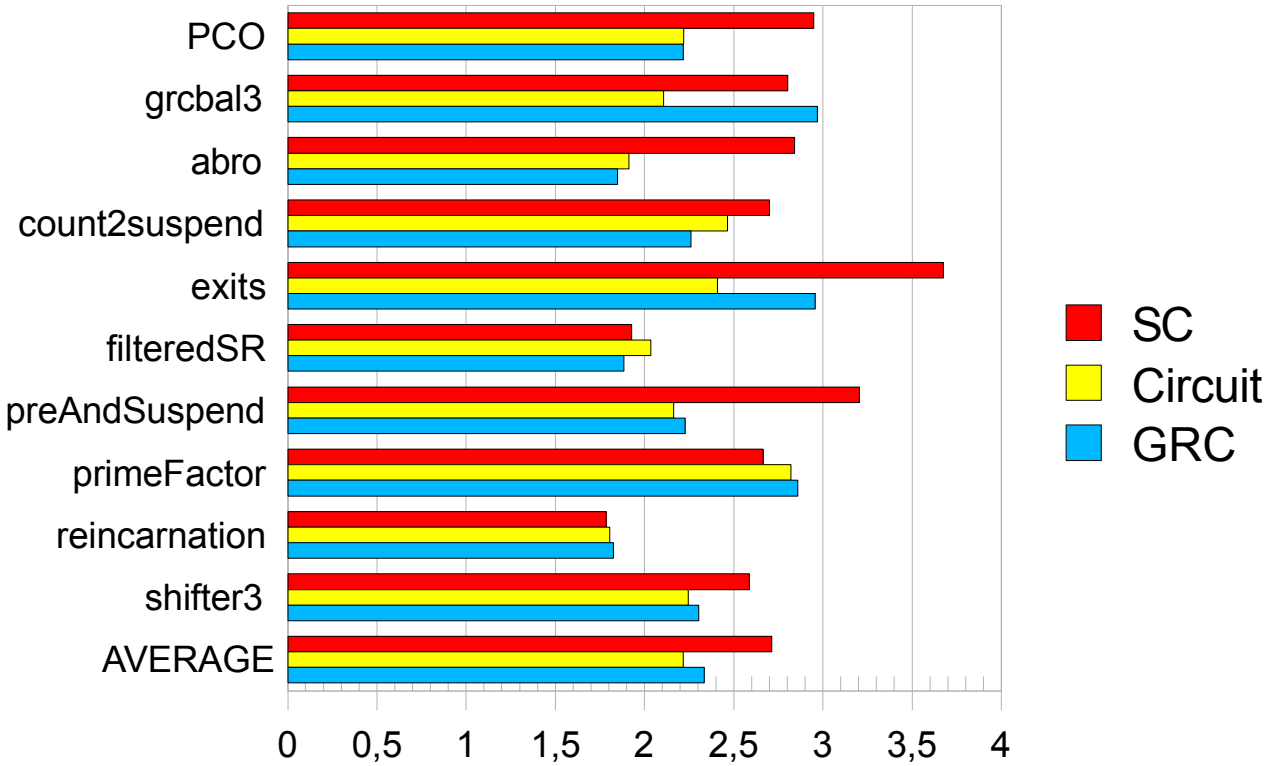
```
==== TICK 0 STARTS, inputs = 01, enabled = 00
==== Inputs (id/name): 0/A
==== Enabled (id/state): <init>
FORK:    1/_L_INIT forks 6/T1, active = 0103
FORK:    1/_L_INIT forks 5/T2, active = 0143
FORK:    1/_L_INIT forks 3/T3, active = 0153
FORKE:   1/_L_INIT continues at TMain
PRESENT: 6/T1 determines A/0 present
EMIT:    6/T1 emits B/1
PRIO:    6/T1 set to priority 4
PRESENT: 5/T2 determines B/1 present
EMIT:    5/T2 emits C/2
TERM:    5/T2 terminates, enabled = 073
PRESENT: 4/_L72 determines C/2 present
EMIT:    4/_L72 emits D/3
PRIO:    4/_L72 set to priority 2
PRESENT: 3/T3 determines D/3 present
EMIT:    3/T3 emits E/4
TERM:    3/T3 terminates, enabled = 017
PRESENT: 2/_L75 determines E/4 present
EMIT:    2/_L75 emits T_/5
TERM:    2/_L75 terminates, enabled = 07
PRESENT: 1/TMain determines T_/5 present
```

Conciseness



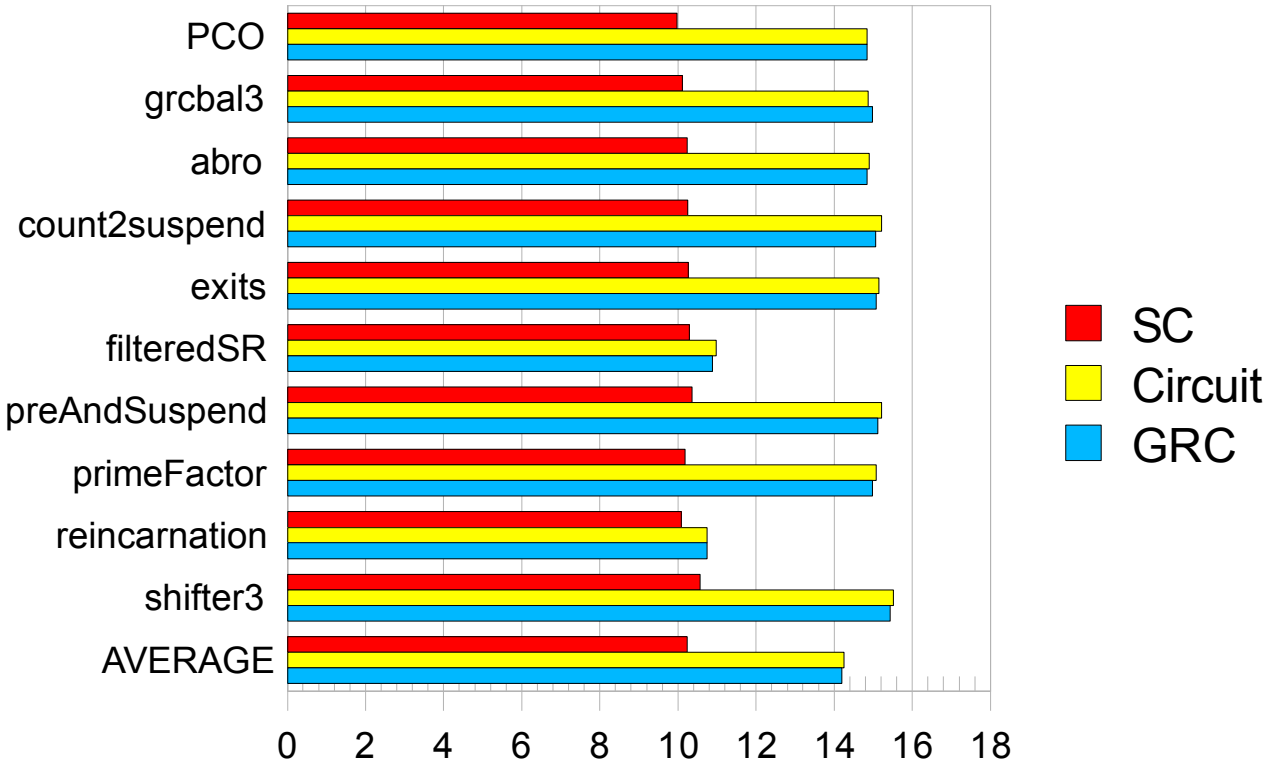
Size of tick function in C source code, line count without empty lines and comments

Code Size



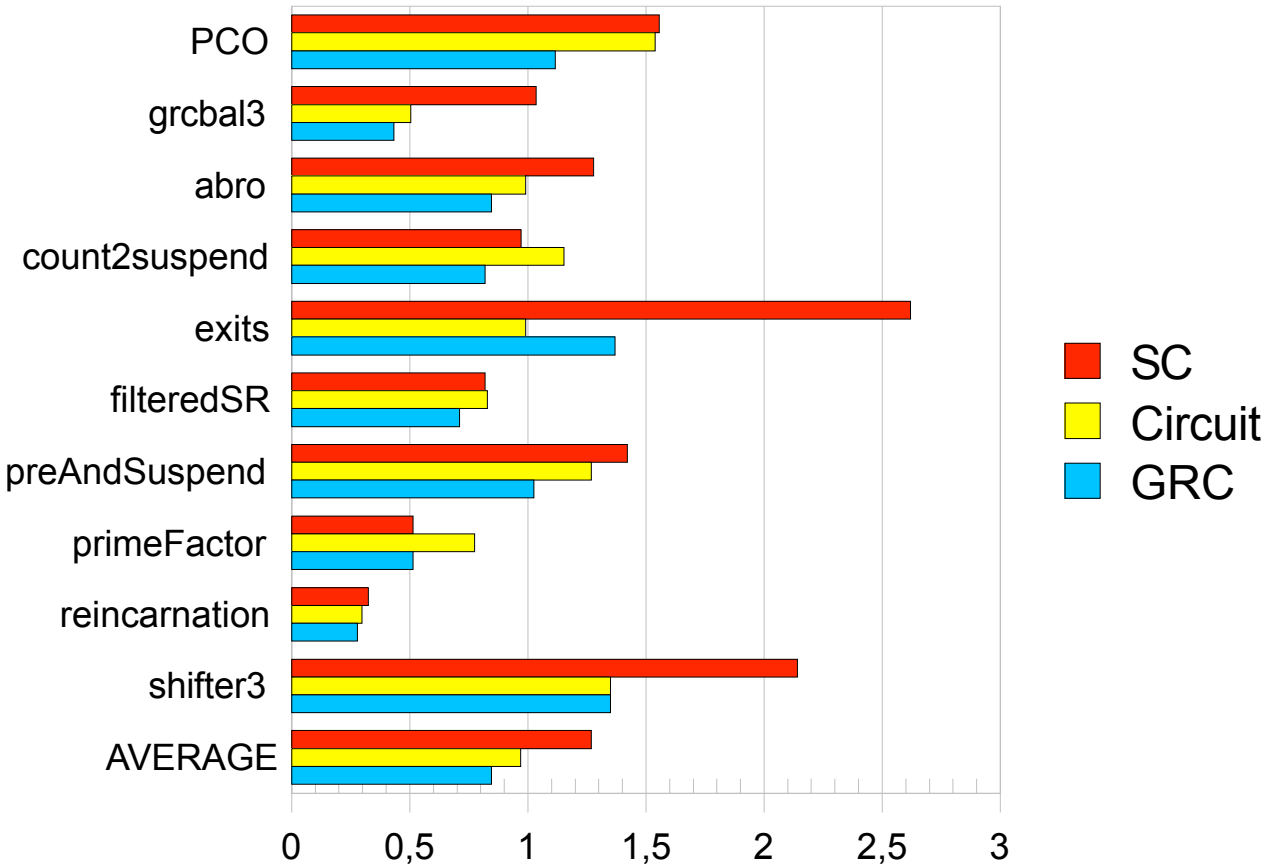
Size of tick function object code, in Kbytes

Code Size



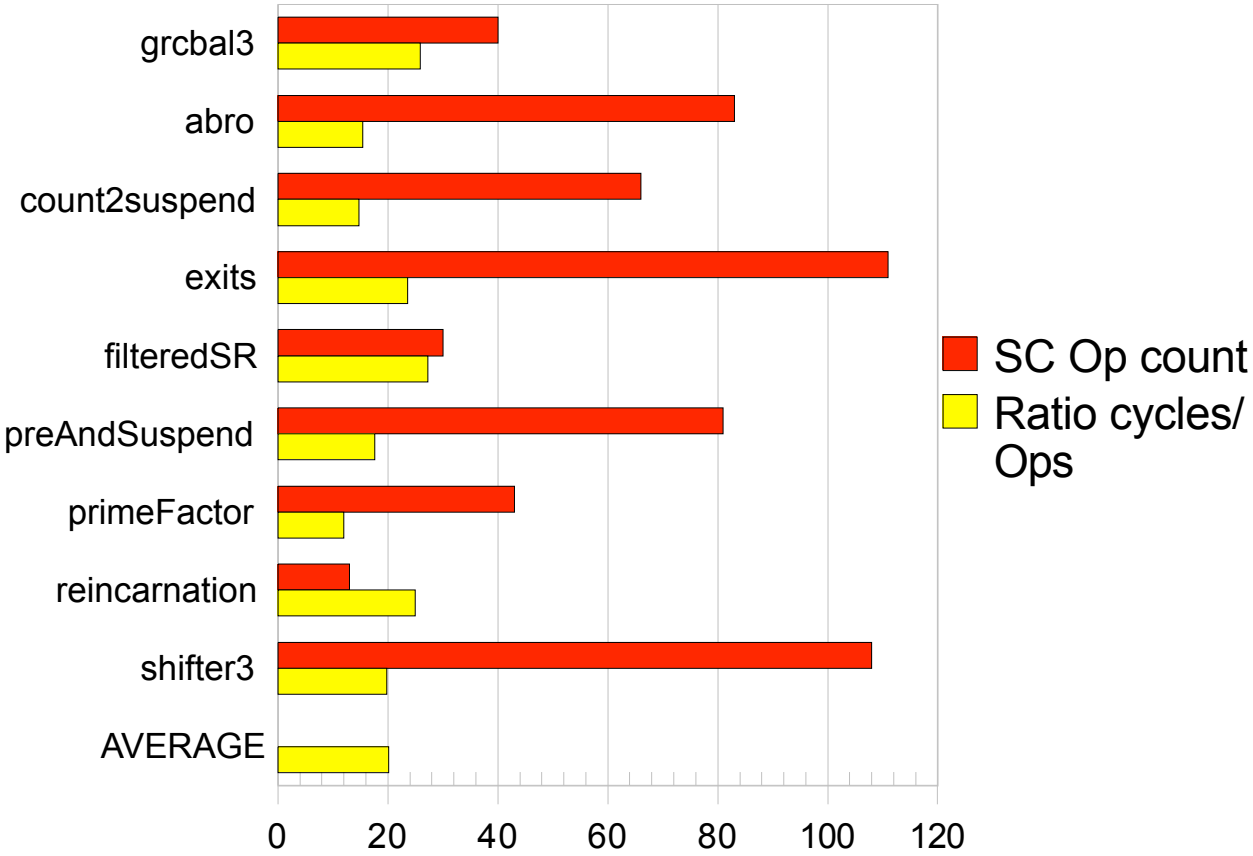
Size of executable, in Kbytes

Performance



Accumulated run times of tick function, in thousands of clock cycles

Operator Density



SC operations count, ratio to clock cycles