Department of Computer & Information Science

Technical Reports (CIS)

University of Pennsylvania *Year* 2006

Network-Code Machine: Programmable Real-Time Communication Schedules (Supplemental Material)

Sebastian Fischmeister [*] Oleg Sokolsky [†]

Insup Lee [‡]

[*]University of Pennsylvania,

[†]University of Pennsylvania, sokolsky@cis.upenn.edu

[‡]University of Pennsylvania, lee@cis.upenn.edu

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-06-01. Initial Paper URL on Scholarly Commons: http://repository.upenn.edu/cis_papers/230

Network-Code Machine: Programmable Real-Time Communication Schedules (Supplemental Material)

Sebastian Fischmeister, Oleg Sokolsky and Insup Lee
Department of Engineering and Applied Science
University of Pennsylvania
sfischme@seas.upenn.edu, {sokolsky, lee}@cis.upenn.edu

Tech. Rep. MS-CIS-06-01

Abstract

This technical report provides supplemental material to the paper titled "Network-Code Machine: Programmable Real-Time Communication Schedules" [2] and contains supplemental data and additional explanations. It is not a standalone work, but should be read together with the original paper.

1 Composite Instructions

For sake of brevity, the original work did not provide the source code of the composite instructions. Here we show, which basic instructions form the composite instruction.

- **Goto.**

```
goto ( jmp ) =
   if ( alwaystrue , #jmp )
```

- **Nop.**

```
nop () =
    if ( alwaysFalse , L0 )
```

- **Signal.** In the code, values *##signal_ch*, *##signal_msgid*, and *##signal_len* are specified in the configuration file.

```
signal ( signal ) =
    if ( isClient , L0 )
    xsend ( ##signal_ch , ##signal_msgid , #signal )
    goto ( L2 )
L0: future ( ##signal_len , L1 )
```

This research is supported in part by NSF CCR-0209024, NSF CNS-0410662, NSF CNS-0509327, NSF CNS-0509143 ARO DAAD19-01-1-0473, ARO W911NF-05-1-0182 and OEAW APART-11059.

```
    goto ( L2 )
L1: receive ( ##signal_ch, #signal )
    halt ()
L2: nop ()
```

- **XSend.**

```
xsend ( ch, msgid, val, loc ) =
    create ( #msgid, #loc )
    send ( #ch, #msgid, #val )
    destroy ( #msgid )
```

- **Wait.**

```
wait ( dl ) =
    future ( #dl, L0 )
    halt ()
L0: nop ()
```

- **FTsend.**

```
ftsend ( ftCh, ch, val, msgid ) =
    if ( #ftChNotEmpty, L0 )
    send ( #ch, #msgid, #val )
L0: nop ()
```

- **Ftreceive.**

```
ftreceive ( ftCh, ch, loc ) =
    if ( #ftChNotEmpty, L0 )
    receive ( #ch, #loc )
L0: nop ()
```

- **Ftasync.**

```
ftasync ( ftCh, duration ) =
    if ( #ftChNotEmpty, L0 )
    goto L2
L0: mode ( usched )
    future ( #duration, L1 )
    goto L2
L1: mode ( sched )
    halt ()
L2: nop ()
```

- **Ftasyncsend.**

```
ftasyncsend ( ftCh, ch, msgid, val, duration ) =
    if ( #ftChNotEmpty, L0 )
```

```
    send ( #ch, #msgid, #val )
    goto L2
L0: mode ( usched )
    future ( #duration, L1 )
    goto L2
L1: mode ( sched )
    halt ()
L2: nop ()
```

- **Ftasyncreceive.**

```
ftasyncreceive ( ftCh, ch, loc, duration ) =
    if ( #ftChNotEmpty, L0 )
    future ( #duration, L1 )
    goto L3
L0: mode ( usched )
    future ( #duration, L2)
    goto L3
L1: receive ( #ch, #loc )
    halt ()
L2: mode ( sched )
    halt ()
L3: nop ()
```

2 Extended Performance Measurements

The interpreter and dispatcher introduce overhead to the running system. To quantify this, we measured the instructions' execution times. Node 1 hosts an Intel Pentium 4 with 1.5 GHz, 512 MB RAM, RTLinuxPro 2.2, and a 3c905C-TX/TX-M [Tornado] (rev 78) with exclusive interrupt access. Node 2 hosts an Intel Pentium 4 with 1.8 GHz, 1GB RAM, RTLinuxPro 2.2, and the same network card and IRQ setup.

Table 1 shows the interpreter's and dispatcher's execution times in nanoseconds. The measurements do not fit a normal distribution, so we only give the median, mode, minimum, and the range. The raw data with all the histograms is available at the web-site.

The column *Count* shows the number of valid measurements. They differ, because some instructions occurred twice in the sampling program and some measurements were missed at the application's manual start up. The column *Median* shows the number designating the ordered set's middle value. The column *Mode* shows the value with the largest number of obervations. The columns *minimum* and *range* provide an overview of the jitter in the system. The high range results from the implementation's prototype status, e.g., we do not turn off interrupts during the execution.

The upper part of the table shows the execution times of individual instructions. Some instructions are missing, because their execution time is similar to some listed ones (e.g., the instruction *mode* is similar to the instruction *handle*). The instruction *nop* shows the overhead introduced by the timing. The statistics show that the interpreter introduces only a low overhead. All instructions have a mode of just a few nanoseconds; we can even subtract 171 or 143 nanoseconds for the timing overhead.

The lower part of Table 1 shows the dispatcher's execution time. The rows $isr@n_x$ show the interrupt service routine's (IRQ) execution time. Whenever the network card receives a packet, this routine executes and stores the arrived packet in the NCM's packet queue. As the statistic shows, this routing usually takes a few microseconds. However, other programs can preempt the ISR and this results the ISR's large

`http://www.seas.upenn.edu/~sfischme/nc` or contact one of the authors

	Count	Median	Mode	Min.	Range
create@n_1	999.964	944	425	419	47.524
create@n_2	999.984	717	366	355	27.455
future@n_1	499.981	380	377	227	27.126
future@n_2	499.990	346	346	191	22.606
handle@n_1	999.964	177	179	173	46.416
handle@n_2	999.984	165	191	146	26.007
if@n_1	499.982	351	348	195	48.847
if@n_2	499.991	331	317	159	21.357
nop@n_1	499.983	174	174	171	24.822
nop@n_2	499.992	150	148	143	21.595
send@n_1	999.964	390	372	371	46.181
send@n_2	999.982	330	315	310	21.748
isr@n_1	998.248	4.349	3.560	3.432	102.804
isr@n_2	998.304	3.983	3.423	3.131	103.511
sendp@n_1	999.964	27.092	22.864	22.227	63.989
sendp@n_2	999.982	25.643	21.887	20.478	43.128

Table 1: The interpreter's and dispatcher's execution times in nanoseconds of n_1 and n_2.

range. Finally the rows *sendp@*n_x show how the execution time of sending a data packet. From similar measurements done for [1], we know that the worst case communication delay is described by the equation: $latency = 0.18979 * datasize + 165$. This value has been identified by an empirical latency test of about one million interchanges for data length of the Ethernet frame starting at size 30 up to the maximum of 1500. The execution times of sending a packet vary according to this equation.

3 Example Program for Raw TDMA Network Access

The following program shows, how to code raw access to TDMA slots. Node 1 and node 2 split up the bandwidth evenly. First at each round, they synchronize their clocks. Then Node 2 is allowed to use the medium for 50 time units. Afterwards, Node 1 has exclusive access. After each TDMA access, we programmed a safety delay of five time units.

Node N_1:
```
L0: signal ( sync )
    wait ( 55 )
    mode ( usched )
    wait ( 50 )
    mode ( sched )
    wait ( 5 )
    goto ( L0 )
```

Node N_2:
```
L0: signal ( sync )
    mode ( usched )
    wait ( 50 )
    mode ( sched )
    wait ( 60 )
    goto ( L0 )
```

References

[1] M. Anand, S. Fischmeister, J. Kim, and I.Lee. Distributed Code Generation from Hybrid Systems Models for Time-delayed Multirate Systems. In *Proc. of the ACM Conference on Embedded Software (EmSoft'05)*, Short paper, 2005.

[2] S. Fischmeister, O. Sokolsky, and I. Lee. Network-Code Machine: Programmable Real-Time Communication Schedules. In *Proc. of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06)*, 2006.