January 2006

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

Sebastian Fischmeister
University of Pennsylvania

Oleg Sokolsky
University of Pennsylvania, sokolsky@cis.upenn.edu

Insup Lee
University of Pennsylvania, lee@cis.upenn.edu

Follow this and additional works at: http://repository.upenn.edu/cis_reports

Recommended Citation

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

This paper is posted at ScholarlyCommons. http://repository.upenn.edu/cis_reports/124
For more information, please contact libraryrepository@pobox.upenn.edu.
Network-Code Machine: Programmable Real-Time Communication Schedules (Supplemental Material)

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.

Comments
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

This technical report is available at Scholarly Commons: http://repository.upenn.edu/cis_reports/124
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.

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

• Nop.

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

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

```plaintext
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.
• XSend.

\[
xsend(\ ch, \ msgid, \ val, \ loc) = \\
create(\ \#msgid, \ \#loc) \\
send(\ \#ch, \ \#msgid, \ \#val) \\
destroy(\ \#msgid)
\]

• Wait.

\[
wait(\ dl) = \\
\text{future}(\ \#dl, \ L0) \\
halt() \\
L0: \ \text{nop}()
\]

• FTsend.

\[
ftsend(\ ftCh, \ ch, \ val, \ msgid) = \\
\text{if (}\ \#ftChNotEmpty, \ L0) \\
send(\ \#ch, \ \#msgid, \ \#val) \\
L0: \ \text{nop}()
\]

• Ftreceive.

\[
ftreceive(\ ftCh, \ ch, \ loc) = \\
\text{if (}\ \#ftChNotEmpty, \ L0) \\
receive(\ \#ch, \ \#loc) \\
L0: \ \text{nop}()
\]

• Ftasync.

\[
ftasync(\ ftCh, \ duration) = \\
\text{if (}\ \#ftChNotEmpty, \ L0) \\
goto \ L2 \\
L0: \ \text{mode (}\ usched) \\
\text{future (}\ \#duration, \ L1) \\
goto \ L2 \\
L1: \ \text{mode (}\ sched) \\
halt() \\
L2: \ \text{nop}()
\]

• Ftasyncsend.

\[
fasyncsend(\ ftCh, \ ch, \ msgid, \ val, \ duration) = \\
\text{if (}\ \#ftChNotEmpty, \ L0)
\]
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, RTLin-
xPro 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 observations. 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@nx 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
Table 1: The interpreter’s and dispatcher’s execution times in nanoseconds of $n_1$ and $n_2$.

<table>
<thead>
<tr>
<th>Function</th>
<th>Count</th>
<th>Median</th>
<th>Mode</th>
<th>Min.</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>create@n(_1)</td>
<td>999.964</td>
<td>944</td>
<td>425</td>
<td>419</td>
<td>47.524</td>
</tr>
<tr>
<td>create@n(_2)</td>
<td>999.984</td>
<td>717</td>
<td>366</td>
<td>355</td>
<td>27.455</td>
</tr>
<tr>
<td>future@n(_1)</td>
<td>499.981</td>
<td>380</td>
<td>377</td>
<td>227</td>
<td>27.126</td>
</tr>
<tr>
<td>future@n(_2)</td>
<td>499.990</td>
<td>346</td>
<td>346</td>
<td>191</td>
<td>22.606</td>
</tr>
<tr>
<td>handle@n(_1)</td>
<td>999.964</td>
<td>177</td>
<td>179</td>
<td>173</td>
<td>46.416</td>
</tr>
<tr>
<td>handle@n(_2)</td>
<td>999.984</td>
<td>165</td>
<td>191</td>
<td>146</td>
<td>26.007</td>
</tr>
<tr>
<td>if@n(_1)</td>
<td>499.982</td>
<td>351</td>
<td>348</td>
<td>195</td>
<td>48.847</td>
</tr>
<tr>
<td>if@n(_2)</td>
<td>499.991</td>
<td>331</td>
<td>317</td>
<td>159</td>
<td>21.357</td>
</tr>
<tr>
<td>nop@n(_1)</td>
<td>499.983</td>
<td>174</td>
<td>174</td>
<td>171</td>
<td>24.822</td>
</tr>
<tr>
<td>nop@n(_2)</td>
<td>499.992</td>
<td>150</td>
<td>148</td>
<td>143</td>
<td>21.595</td>
</tr>
<tr>
<td>send@n(_1)</td>
<td>999.964</td>
<td>390</td>
<td>372</td>
<td>371</td>
<td>46.181</td>
</tr>
<tr>
<td>send@n(_2)</td>
<td>999.982</td>
<td>330</td>
<td>315</td>
<td>310</td>
<td>21.748</td>
</tr>
<tr>
<td>isr@n(_1)</td>
<td>998.248</td>
<td>4.349</td>
<td>3.560</td>
<td>3.432</td>
<td>102.804</td>
</tr>
<tr>
<td>isr@n(_2)</td>
<td>998.304</td>
<td>3.983</td>
<td>3.423</td>
<td>3.131</td>
<td>103.511</td>
</tr>
<tr>
<td>sendp@n(_1)</td>
<td>999.964</td>
<td>27.092</td>
<td>22.864</td>
<td>22.227</td>
<td>63.989</td>
</tr>
<tr>
<td>sendp@n(_2)</td>
<td>999.982</td>
<td>25.643</td>
<td>21.887</td>
<td>20.478</td>
<td>43.128</td>
</tr>
</tbody>
</table>

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: \( \text{latency} = 0.18979 \times \text{datasize} + 165 \). 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\(_1\):**

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

**Node N\(_2\):**

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

### References