A Mobile Music Environment Using a PD Compiler and Wireless Sensors

Robert Jacobs, Mark Feldmeier, Joseph A. Paradiso

MIT Media Lab

rnjacobs@mit.edu, carboxyl@mit.edu, joep@media.mit.edu

Abstract

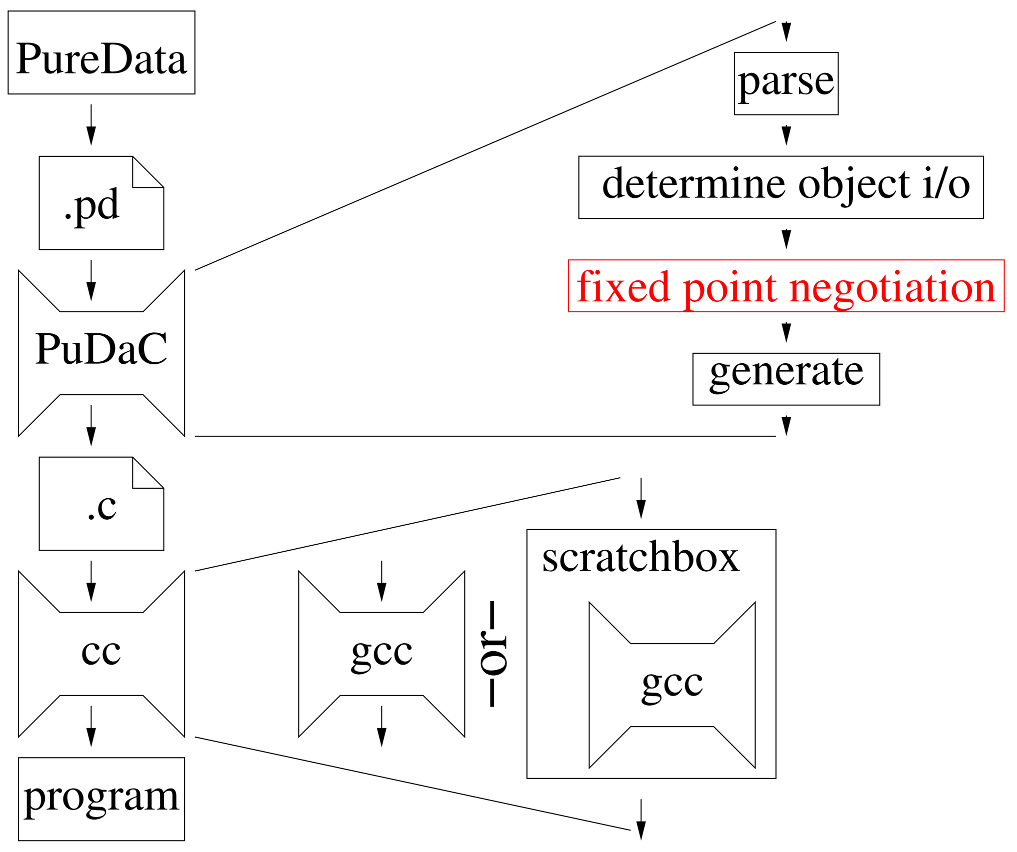
We describe a framework for a wireless sensor-based mobile music environment. Most prior work in this area has not been truly portable, or has been limited to simple tempo modification or selection of pre-recorded songs. The exceptions generally focused on external data rather than dynamic properties and states of the listener. Our system exploits a short-range wireless sensor network (using the ZigBee protocol and inertial sensors) and a compiler for PureData, a graphical music processing language. We demonstrate the system in an interactive exercise application running on a Nokia N800.

What about PDa?

PureData Anywhere[1]'s goal is different – PDa was specifically designed to enable the full PureData interaction mode and functionality on handheld FPU-less devices. PuDaC, however, trades the interactivity for greater optimization of the entire system.

Compiler

Our PD compiler presents a middle ground between PD and a lower-level language like C, with much of the ease of use of PD and much of the speed of C. By using a highly optimizing C compiler, many of the inefficiencies due to mechanical translation are further eliminated. For example, many objects in PD patches have exactly one incoming connection. A good C compiler, if told to optimize sufficiently, will take these objects and put them inside their callers. Further optimizations would then go and find redundant checks on the data type of the incoming message and discard the second set of checks.



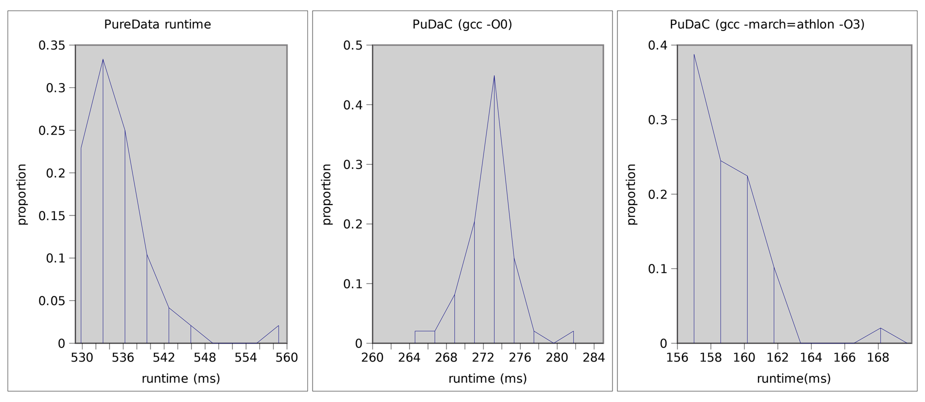
Since we are converting between one format to another of text, we have written the compiler in Perl. Perl excels at parsing text, especially rigidly defined text like PD's save format. After parsing the entire file into a structure in memory, we execute the generators for all the objects that produce the C code for each.

The compiler takes in a plain text PureData patch file and produces C output. This allows us to take advantage of the large amount of work that other people have put into optimizing compilers without having to implement it ourselves.

The PureData Compiler (PuDaC) replaces each object with a uniquely-named subroutine (and possibly some uniquely-named globals). Each "wire" connecting objects is replaced with directly calling the connected object.

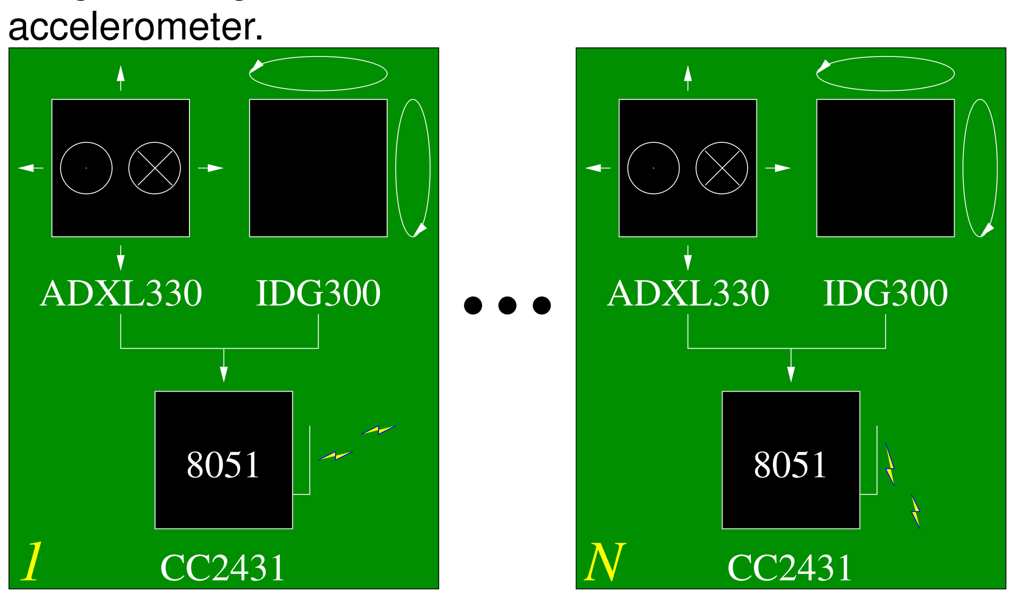
The compiler runs in two passes (three if you include the additional stage of running gcc). First it parses the input file, loading all the objects into an associative array with a UID for the object as the key. The value of each entry is an another associative array with several predefined entries specifying the object's name, the C representation of the objects attached to the inlets and outlets of the object, the C-generating perl code for this object, and a redirection specifying the object has another name (like `sel / select`). Then it prints a prologue, executes the C-generating perl code for all objects, and prints the main function. This model makes debugging tremendously easier, although it is probably significantly less efficient than is possible.

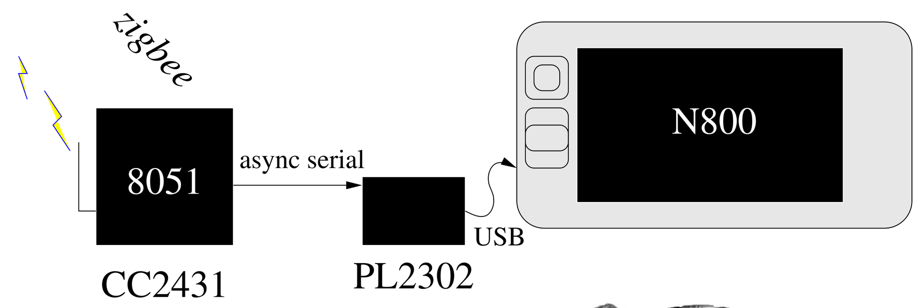
Even so, a simple test patch (which does one million floating point multiplies) shows a significant performance increase over the plain interpreter: on an Athlon (Thunderbird core) running at 1066 MHz, PureData takes an average of 533ms, compared to as little as 158ms for the optimized version of the compiler output, about 30%. (Histograms of trials are below.)

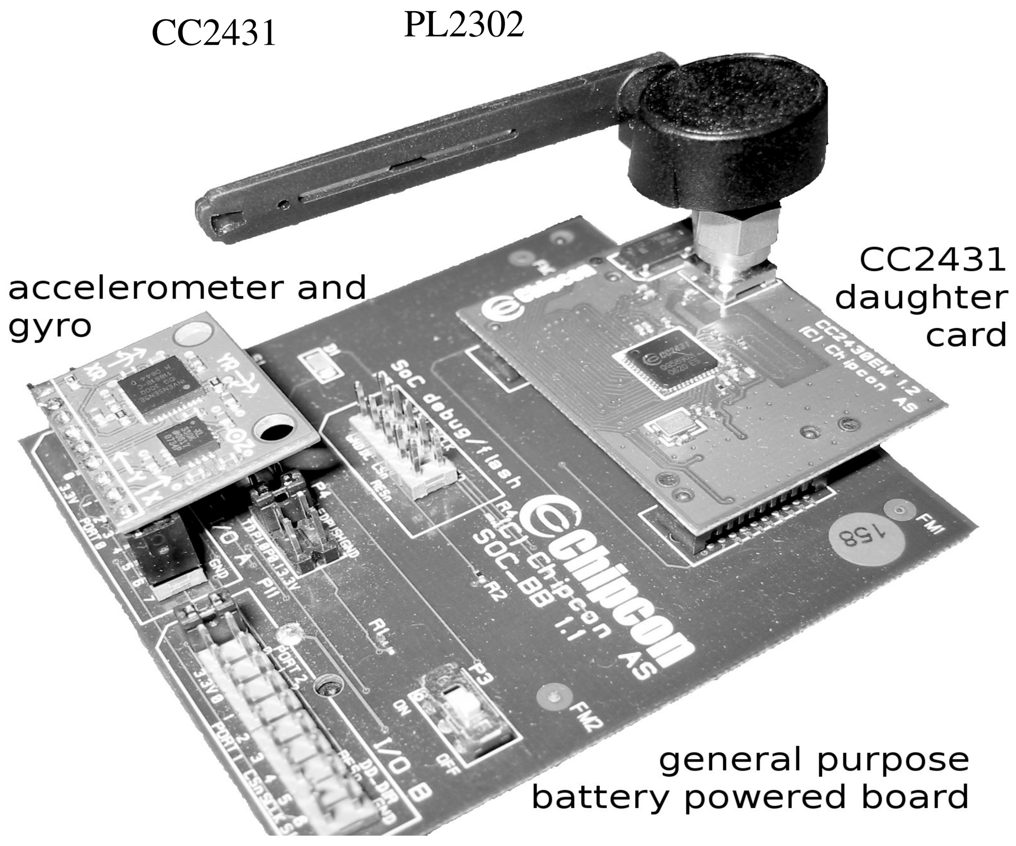


Physical Instantiation

The sensor system looks very straightforward. Each sensor should use a microcontroller, a simple radio, and an accelerometer, preferably three-axis. When we were introduced to the CC2431, it looked like an ideal solution, because it had almost everything needed already in it. It's tolerant of a wide range of voltages and so could be run directly off a battery, contains an ADC, and contains its own integrated ZigBee radio. All we had to add would be the accelerometer.



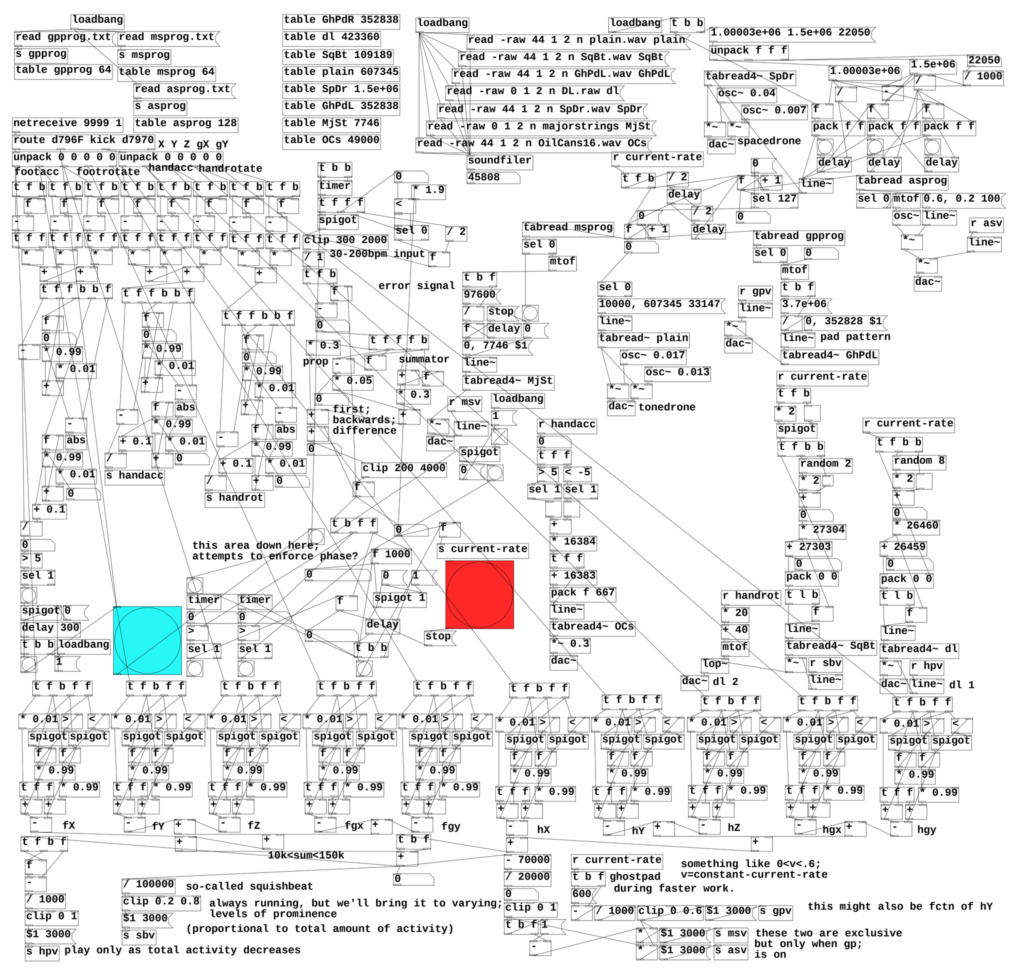




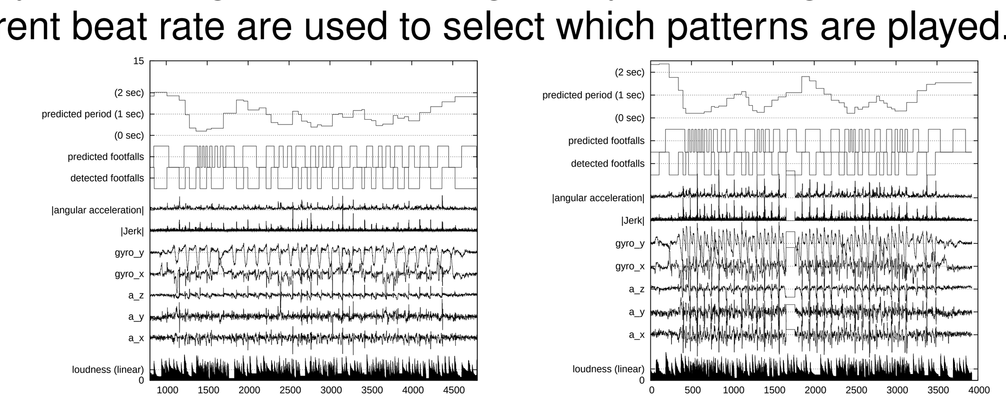
The 5-axis accelerometer/gyroscope ("IMU", Inertial Measurement Unit) board from SparkFun Electronics contains an Analog Devices ADXL330M and an InverSense IDG300. The ADXL330M measures $\pm 3.6\ g$ on all 3 axes, and is configured to give a 50Hz bandwidth. The IDG300 measures $\pm 500°/s$ about the two axes not normal to the chip, and has a 140Hz bandwidth.

Nokia's N800 is a small portable computer (measuring 2.95 x 5.66 x 0.51 inches, weighing 7.26 ounces). It includes bluetooth, 802.11g, an ARM11 processor, a dedicated DSP, and runs the Linux-based Maemo internet tablet software suite. It shares general features with similar handheld computing devices, so the exact choice of platform is flexible.

Mapping



One mapping patch is shown above. The data is accepted, routed, and the acceleration and angular velocity from the IMUs are converted to jerk and angular acceleration. We then compute the ratio of local mean to local average deviation, look for when that exceeds a given threshold, and feed this output into a delay with holdoff. We also compute the local maximum and minimum of each axis, and subtract to get a local dynamic range. The delay with holdoff feeds into a phase-locked loop (PLL) that attempts to match the phase and frequency of the input, which is attached to simple logic that runs several audio patterns. The dynamic range, rate of change of dynamic range, and current beat rate are used to select which patterns are played.



An example showing how well it works can be seen above. The Z axis of the accelerometer (labeled a_z) was oriented normal to the leg, in the direction of stride (because no rotation was expected about this axis). The X axis (a_x) was parallel to the leg. |Jerk| and |angular acceleration| were calculated as the magnitude of the first backwards differences on available axes. The detected and predicted footfalls plots indicate a detection or prediction when they change from low to high or vice versa. The period ranges from 0 to 2 seconds. Both plots are approximately 40 seconds (4000 centiseconds) long. The gap seen from 16-17 seconds in the bottom graph is because the data dumping program momentarily paused.

For jogging data, both the gyroscope and accelerometer data provided a good impulse source, and the detected footfalls plot shows this. Unfortunately, as can be seen in both plots, the predicted period oscillates with a period of 10 steps, never successfully really getting to the true pace. This is solely the fault of the PID (proportional integral derivative) controller inside the phase-locked loop. Later tuning should make this react better, but it is hard to adjust well.

Conclusions

Because the N800 has a floating point unit, we did not look into implementing automatic fixed point casting of the various numbers. As such, the compiler is not yet very useful on small hardware such as cell phones or similar devices, but the small size of the N800 and functional equivalents compares favorably with current portable audio players. Additionally, several PD functions (such as `cos~` and `osc~`) are implemented using the FPU; for machines that lack a FPU, a lookup-table based solution will be necessary.

The compiler is distinctly to the point where it produces useful results, but it needs a lot more effort to bring it to the point where it could be used in a commercial setting. It has a number of rough edges, but can now be used on an experimental basis for further development.

References

[1] G. Geiger. PDa: Real time signal processing and sound generation on handheld devices. In *Proceedings on the International Computer Music Conference*, 2003.

NIME 2008, New Interfaces for Musical Expression, 5 - 7 June 2008, Genova, Italy