High-Level Languages for Small Devices: A Case Study

Manuel Carro
School of Computer Science
U. Politécnica de Madrid
mcarro@fi.upm.es

José F. Morales
Facultad de Informática
U. Complutense de Madrid
jfmorale@fdi.ucm.es

Henk L. Muller
C.S. Department
University of Bristol
henkm@cs.bris.ac.uk

G. Puebla
School of Computer Science
U. Politécnica de Madrid
german@fi.upm.es

M. Hermenegildo
U. Politécnica de Madrid and
U. of New Mexico
herme@{fi.upm.es,unm.edu}

ABSTRACT

In this paper we study, through a concrete case, the feasibility of using a high-level, general-purpose logic language in the design and implementation of applications targeting wearable computers. The case study is a "sound spatializer" which, given real-time signals for monaural audio and heading, generates stereo sound which appears to come from a position in space. The use of advanced compile-time transformations and optimizations made it possible to execute code written in a clear style without efficiency or architectural concerns on the target device, while meeting strict existing time and memory constraints. The final executable compares favorably with a similar implementation written in C. We believe that this case is representative of a wider class of common pervasive computing applications, and that the techniques we show here can be put to good use in a range of scenarios. This points to the possibility of applying high-level languages, with their associated flexibility, conciseness, ability to be automatically parallelized, sophisticated compile-time tools for analysis and verification, etc., to the embedded systems field without paying an unnecessary performance penalty.

Categories and Subject Descriptors

D.1.6 [**Programming Techniques**]: Logic Programming; D.3.2 [**Programming Languages**]: Language Classifications—*Constraint and Logic Languages*; D.3.4 [**Programming Languages**]: Processors—*Compilers, Optimization, Code Generation*; B.1 [**Hardware**]: Control Structures and Microprogramming—*Microprogram Design Aids*

General Terms

Languages, Performance

Keywords

Optimizing compilation, wearable computers, program analysis and transformation, (constraint) logic programming

1. INTRODUCTION

In recent years software has become truly ubiquitous: a large part of the functionality of many devices is now provided by an *embedded* program, which often implements the core tasks that such devices perform. This includes from simple timers in ovens or fuzzy logic based monitoring and control software in household appliances, to sophisticated real-time concurrent systems in cars and cell phones. Upcoming *wearable computing* applications envision an integration of such devices even into clothing.

A range of micro-controllers is available for these purposes which, when compared with the processors currently used in workstations or laptops, are much less expensive and consume a reduced amount of power (starting at micro-Watts for the simplest ones). In return such processors have limited memory (from hundreds of bytes to perhaps a few megabytes total) and speed (up to at most a few hundred megahertz clock rates, and with little or no instruction parallelism). Basically, lower clock rates consume less power and simpler processors with less storage are cheaper.

As a result of this, frequent requirements on *embedded programs* is that they be able to use minimum storage, execute few instructions, and meet strict timing constraints, since all this brings down both cost and power consumption. The importance of these requirements depends of course on the domain. Because of these requirements, programs are often developed in low-level languages including, in many cases, directly in assembler [30]. Some of those programs are written on micro-controllers in order to completely minimize power consumption while others are written using also small, but more general-purpose computing platforms [15, 17, 22]. In most cases, platform limitations drive the whole development cycle, diverting attention from modularity, reusability, code maintainability, etc.

At the same time, and despite resource and program development technology constraints, the functionality implemented by embedded systems is often quite sophisticated. This can include, even for the smallest devices, non-trivial matrix operations (as in, e.g., Kalman filters [29], used in GPS receivers), or intensive, real-time operations on data

streams (including spatialization, as in the digital sound processing example that we will study in this paper). In addition, more sophisticated functionality and more automated operation is always demanded by users. Furthermore, those systems often face strict correctness requirements because of the nature of the application or simply because of the higher cost of fixing bugs once the system is deployed. In practice, and in order to deal with these conflicting requirements, applications are often coded also in a high-level or specification language which is used for prototyping and verification, in addition to the above mentioned low-level language, which constitutes the implementation. Unfortunately, often no real link between these two codings of the problem exists.

Coding at a Higher Level

A number of recent proposals make it easier to code stream processing routines. They are usually based on connecting processing blocks (available in a library or provided by the user) using a textual programming language (e.g. [25]) or visual depictions thereof (e.g. [12]). In many cases their abstraction level is adequate to use them as specification languages, but code generation is sometimes not automatic, or the resulting code needs to be fine-tuned by hand. Data and control models are often that of a procedural / O.O. language, which makes the application of some program analysis and transformation techniques somewhat challenged. Domain-specific program transformation techniques exist, but they have only a limited use in the case of a general embedded system. We want to note that defining processing blocks and applying domain-specific transformations is in principle possible for languages of any type and level.

In contrast, the availability of optimizing technology for high-level languages makes their direct use to implement (and not just to specify and to prototype) an attractive alternative. First, using high-level languages makes it easier to write better programs, with fewer errors, in less time, and with less effort. Problems can be formulated at a higher level of abstraction and much of the low-level detail that must be dealt with when using, e.g., C or assembler (such as manual memory management, ensuring safe typing, complex data structure management, etc.), which complicate and obfuscate the coding of algorithms, are taken care of automatically. These languages also make it easier to detect any remaining bugs and also to verify the correctness of programs automatically. Finally, high-level languages are also useful in the context of the general trend in processor design towards multi-core chips. Dual processor designs (with four threads total) are present already in mainstream laptops and the expectations are to double the number of cores and threads every two years at fixed cost. Since the motivation behind these multi-core designs is precisely to gain performance while keeping resource consumption down, this trend is also likely to hit the micro-controller arena. Parallelized programs will be required to exploit the performance that the chip can deliver, and the parallelization task will add to the burden on the programmer. High-level languages are relevant in this context because they have been shown to be easier to parallelize automatically [8].

The challenge in using high-level languages in embedded and wearable devices is to be able to generate automatically executables that are as efficient as required by the platform (with memory, speed, and energy consumption close to hand-coded low-level implementations). A particular challenge is to achieve this even if numeric or data-intensive computations are involved. While some interesting work has been done regarding the use functional programs in embedded systems [19, 28], the use of (constraint) logic programming (CLP) systems in this context has received comparatively little attention. CLP, and, in particular, the availability of logical variables, search, and constraints in a programming language can be attractive because these features can make it easier to provide sophisticated problem solving, optimization, and reasoning capabilities in devices. This is in line with the demands for higher and more automated functionality from users. The purpose of this paper is to investigate for a particular case study (a sound spatializer embedded in a wearable computer) the feasibility of coding it using a very high level, multiparadigm programming system supporting predicates, logical variables, dynamic typing, search, and constraints in combination with functions, higher order, objects, etc. (in particular, the Ciao system [4, 9, 10]).

However, the point of the paper is not to use all these capabilities extensively[1] but instead to study whether current state of the art tools for compile-time analysis, verification, specialization, and low-level optimization are powerful enough to optimize away the default functionality available in such a rich language, including all its libraries, for a program such as the spatializer which only needs a fraction of them. This will require optimizing away all the overhead needed for supporting backtracking, full unification, tagged values, infinite precision arithmetic, etc., which are present by default in the language *for program sections that do not need these features* and see whether it is possible to produce in this way executables for the wearable computer that are competitive in terms of speed, memory consumption, etc., when compared to a solution in a low-level language (in our case, C). This presents challenges that, while having some similarities, are also different for example from those which appear when optimizing programs written in other languages: dealing with logical variables and argument modes (i.e., procedure arguments are not known *a priori* to be input or output), dealing with backtracking and multiple solutions, eliminating dynamic typing (when compared to strongly typed languages), etc.

A Concrete Problem and its Motivation

The case study we chose is a stylized (but fully functional) version of a real wearable computing application (designed for the new Bristol CyberJacket) in which a set of virtual sounds are projected into a physical space. The user experiences these *soundscapes* through a set of headphones attached to the wearable computer (which has limited available power). An example of the use of such a *sound spatialization* device is a "talking museum" where any object, from the actual exhibits to the walls or doors of the rooms, can appear to be talking to the visitor. A compass is fixed on the user headphones which provides information on head orientation. The wearable computer is also aware of the user's location, through GPS for outdoor locations and through an

[1] A brief account of how CLP characteristics can be of use to define and implement processes on streams appears in [24]. Our work is complementary in that we do not deal with how to define and compose basic building blocks, but rather on how to optimize them.

Figure 1: Sound spatializer prototype, with Gumstix (bottom left) and compass (right) attached to headphone.

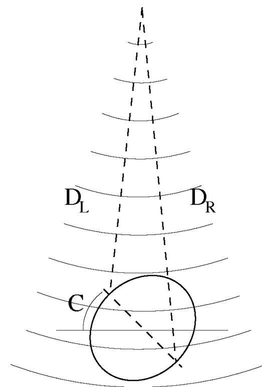

Figure 2: Sound samples reaching the ears.

ultrasonic positioning system [15] for indoor installations. With these two sources of information the wearable device can determine where a sound should be positioned relative to the user. By calculating the angle at which the sound is with respect to the head, the delay that the sound will experience at each ear can be calculated, and this allows spatializing the sound [2]. For the sake of simplicity, and since we want to show actual code, we will present a version in which position is not dealt with, and only sound direction is taken into account.

This concrete case study was selected because of its characteristic nature: it requires core functionality present in many wearable computing applications. Handling streams of data such as audio and video and collections of positions is frequent in pervasive and wearable systems. In many common scenarios one or more sensors will produce data streams to be received and used by an actuator. These sensors can generate data at different, unrelated, but generally fixed, and sometimes very high, rates. Additionally, this case does not belong to the restricted class of synchronous systems and the operation (and, therefore, time) of some of the actuators depend on the particular data coming in. Therefore, this case study exemplifies a family of programs to which techniques similar to those we will show here can be applied. Very often (including, for example, our case) these problems have, in addition to resource constraints, hard real-time constraints where there are exact deadlines within the system. Of course the objective is to be able to support, in addition to such lower-level data integration tasks, higher-level functionality. But the point of the study is to see if the lower-level tasks can be handled efficiently enough, since the suitability of the programming language used for the higher-level tasks is taken for granted.

2. THE SOUND SPATIALIZER

The problem we focus on is spatializing sound in real time by processing a monaural stream into a stereo one so that the sound appears to come from a position in space when played through a set of headphones. Angle information comes from a compass mounted on the headphones. When the head turns, the compass will register a change in heading and the spatialization unit should change accordingly the direction from which the sound seems to originate to create the illusion that it remains fixed at a certain spacial point. Our fully functional prototype has a small processor board, compass, and battery, all integrated on a pair of headphones (see Figure 1). The sound stream is a series of samples (16-bit integers, coming either from some external source or from flash memory) and the compass data is read as floating-point numbers measuring the heading in degrees relative to North.

We will assume that signals are delivered at *a priori* known rates, and we will apply analysis and optimization tools in order to reduce the resources (processor cycles, mainly) needed to deliver sound in a timely manner. We want to, at least, be able to execute spatialization in real time on a small processor (described in Section 2.3). Any gains beyond real-time execution will allow us to lower the clock rate of the processor, which reduces power consumption, in turn increasing battery life. In the following subsections we will discuss the requirements in detail.

2.1 Sound Spatialization Basics

Figure 2 sketches how the ear localizes sound emanating from some point in space. When the head does not face the sound source, sound waves travel a different distance to each ear (D_L and D_R, respectively). Therefore the left and right ears receive the same sound with a slight phase shift, in the order of a millisecond, determined on the basis of the difference $D_L - D_R$. This enables the brain to determine the direction of the sound. Calculating that shift is the starting point for spatialization, as each earphone is to output one stream of sound samples, which is in turn a possibly delayed copy of the initial sound stream. The *absolute* distances to the sound sources can also be used to modify the volume of the sound, although in practice attenuation information is hardly used by the brain when determining the source of a sound. $D_L - D_R$ obviously depends on the angle C and the size of the head.

2.2 Sound Quality and Spatial Localization

High sampling rates are needed to model small head movements. A relative displacement of 3.43 mm. corresponds to a difference of 10 μs, which needs a sampling rate of 100 KHz. The higher the sampling rate, the better the spatialization, but the more processing is required: there is a trade-off between quality of spatialization and processing power.

One of the requirements of the final application is that CD-quality sound has to be produced, i.e., 44,100 16-bit samples per second (sps), which can model a relative displacement between ears of about 7.5 mm. (a rotation of 2

degrees.) Our program should therefore be able to process 44,100 16-bit sps and deliver 88,200 16-bit sps (on the two output channels). Concurrently, data from the compass has to be read and used to produce the sound streams.

2.3 Hardware Characteristics of the Platform

The target architecture is modest in comparison with modern desktops or laptops: it is a Gumstix board equipped with a 200MHz XScale processor, 64Mb of RAM and 4Mb of flash memory, which acts as permanent storage, and running a version or Linux (see `http://www.gumstix.com/` for more information). The Gumstix board is around 25 times slower (depending on the application, of course) than a 1.5GHz SpeedStep Centrino, at a fraction of the power usage. Memory and storage limitations are obviously significant and relevant to our application, since we want to run a non-terminating process, and thus garbage collection is critical.

2.4 Hard Real-Time

A stereo sample should ideally be generated every 22 μs (1/44,100), as input samples arrive. In practice, sound buffers require blocks (typically of 256 samples) to be written to the sound card. Thus we are required to produce a block of sound samples every 6 ms. Two issues can prevent us from meeting this hard real-time deadline: process scheduling may swap out our program for more than 6 ms. and executions of the garbage collector could take more than that.

The former can be worked around, if necessary, by switching to some form of real-time Linux. However, if fewer processor cycles are needed by the application (our goal), it will be less likely to be affected by the O.S. scheduling. In our case this proved not to be a problem in the end.

The latter could hinder the use of high-level languages, as automatic memory management is one of the characteristics which makes them less error-prone. Some languages have undergone a careful and interesting revision of the memory management model and primitives in order to adapt to real-time requirements [3] (but, arguably losing part of the initial elegance). In our case recent work in this regard is aimed at inferring bounds on memory consumption at compile-time in order to guarantee compliance with memory constraints without modifying the language. In any case in our concrete case study the built-in memory management of the (Ciao) system proved sufficient to not cause any noticeable interruption in the sound while keeping memory consumption constant and within reasonable limits.

In both cases, increasing the sound card buffer size would give more freedom to the application. However, this creates a lag between the compass movements and the sound emission which would render the application unacceptably sluggish and destroy the illusion of spatialization.

2.5 Compass and Concurrency

Reading the compass data (which is done from a serial interface) may take an unknown amount of time, because due to limitations of the hardware data may be corrupted. In order not to block or obfuscate the rest of the application code, a separate thread is started which asynchronously reads data from the compass and posts it for the main program. Communication is performed via an atomically updatable, concurrent dynamic database [5]. This isolates low-level details of the compass from the rest of the program.

```
mono := new InputPeriodicStream(sound_sps);
direction := new InputPeriodicStream(compass_sps);
stereo := new OutputPeriodicStream(sound_sps);
while (true) do
        state = f(mono.current(),
                  direction.current(),
                  state);
        stereo.output(state);
end
```

Figure 3: Single-loop algorithm for the spatializer.

```
mono := new InputPeriodicStream(sound_sps);
direction := new InputPeriodicStream(compass_sps);
stereo := new OutputPeriodicStream(sound_sps);
while (true) do
        state := f_c(direction.current(), mono.current(), state);
        samp_sound := sound_sps/compass_sps;
        while (samp_sound > 0) do
                state := f_m(mono.current(), state);
                stereo.output(state);
                samp_sound := samp_sound − 1;
        end
end
```

Figure 4: A nested-loop sound spatializer.

However, it makes it necessary for the analysis tools to understand this communication by giving them an appropriate description. Ciao includes an assertion language which was used to annotate the interface to the compass (Section 4.2) appropriately.

Note that the scheduling is handled by the Gumstix operating system. In other scenarios, CLP-based tools have shown their usefulness at precomputing feasible schedulings using system specifications written as logic programs which are then automatically specialized to reduce or eliminate scheduling overhead [14].

3. PROGRAM CODE AND SOURCE-LEVEL TRANSFORMATIONS

3.1 Naive Implementation

A naive implementation of the sound spatialization algorithm is shown in Figure 3. A function f takes the current samples of the sound stream and the direction stream, and produces a stereo sample. We encapsulate knowledge about when to skip samples and any history needed in a separate object "state," making f a pure function. The three stream objects all have preset periodicities and are initialized with their expected sampling rates.

This code is naive in that inside the function f one needs to perform trigonometric functions, but these only need to be executed once every compass poll (in our case, once every 4,410 sound samples instead of every sample). In general, a function f that operates on n inputs $s_0, s_1, \ldots, s_{n-1}$ can be projected onto a series of functions $f_0, \ldots, f_{n-1}$ such that

$$f(s_0, s_1, \ldots, s_{n-1}) = f_0(s_0, f_1(s_1, \ldots f_{n-1}(s_{n-1}) \ldots))$$

If the s_i are ordered according to their update rates so that s_0 has the fastest one and s_{n-1} has the slowest one, the initial program can be rewritten to save the results of function applications by computing $f_{n-1}(s_{n-1})$ in the outer loop (with the lowest frequency) and proceeding inwards across nested loops until $f_0(s_0, \cdot)$ is computed in the innermost

```
spatialize(SamplesRemaining, SampleL, SampleR, CurrSkip):-
   new_sample_cycle(SamplesRemaining, NewCycle,
                    CurrSkip, NewSkip,
                    SampleL, SampleR,
                    NewSampleL, NewSampleR),
   new_sample(NewSampleR, R, RestSampleRight),
   new_sample(NewSampleL, L, RestSampleLeft),
   play_sample(R, L),
   spatialize(NewCycle, RestSampleLeft,
              RestSampleRight, NewSkip).

new_sample_cycle(0, ~audio_per_compass, CurrSkip,
                 NewSkip, SL, SR, NSL, NSR):-
   find_skip(~read_compass, NewSkip),
   skip(NewSkip - CurrSkip, SL, SR, NSL, NSR).
new_sample_cycle(Cycle, Cycle - 1,
                 Sk, Sk, SL, SR, SL, SR):- Cycle > 0.

new_sample([Sample|Rest], Sample, Rest):-
   var(Sample) -> read_sample(Sample) ; true.
```

Figure 5: Main loop for the sound spatializer reading from a compass.

loop. Note that in our example code we only deal with the case in which the two frequencies divide each other. This is not the case for arbitrary sensors.

The code for the sound spatializer, according to this decomposition, is shown in Figure 4. The function f has been decomposed into f_m and f_c, and two loops have been created. The outer loop computes f_c when a new compass signal is available, whereas the inner loop applies f_m at a higher frequency. More efficiency is attained at the cost of a slightly more complex code (which has however a clear structure) and the decomposition of f.

3.2 High-Level Code for the Sound Spatializer

To go from the schematic code to a full implementation in a low-level imperative language requires quite a bit of coding where, e.g., memory management (allocation and management of buffers), data types and sizes, explicit synchronization, etc. need to be taken into account. Given our objectives, instead we wrote a complete sound spatializer in Ciao whose *actual* core code is shown in Figure 5 (we do leave out however for brevity some low-level details that deal with obtaining compass data and sending audio data, which were notwithstanding fully implemented in the code which was benchmarked in this paper). Note that while the code has of course to deal with some low-level details, such as actually reading stream information and outputting sounds, there are many others (such as internal buffer information, types and precision of variables, etc.) which do not need to be explicitly stated.

A Note on Syntax: Ciao allows the use of functional notation with no execution time penalty [6]. The prefix operator `~` enables the use of a predicate as a function by making its last argument correspond to the function result. Hence, the goal `?- append([1], [a], R).` can be written as `?- R = ~append([1], [a]).` Predicates can also be defined in functional syntax, by using `:=` instead of `:-` (Figure 6). This assumes that the last argument will represent the function result. Arithmetic expressions are also translated.

The sound stream is represented as an open-ended (incomplete), unbound-length list of samples (of some opaque type) which is incrementally instantiated as more samples

```
sound_sps    := 44100.      % Samples per second
compass_sps  := 10.         % Samples per second
sound_speed  := 343.        % Meters
head_radius  := 0.1.        % Meters
pi           := 3.141592.

audio_per_compass :=
    integer(~sound_sps / ~compass_sps).

samples_per_meter :=
   ~sound_sps / ~sound_speed.

ear_dif(Angle) :=
   ~head_radius * sin((Angle * ~pi) / 180).

find_skip(Angle) :=
    round(~samples_per_meter * 2 * ~ear_dif(Angle)).
```

Figure 6: Physical model in the sound spatializer.

are needed. This list is held in memory and the unnecessary items (the samples which have already reached the farthest ear and are unreachable in the program) are eventually and automatically deallocated.

On the other hand, the compass is explicitly polled (this is the functionality offered by the hardware) by a separate thread and communicated through the predicate `read_compass/1` which returns the latest read value. Based on it, `find_skip/2` determines the current difference (in number of samples) between the left and the right ear. This is used by `skip/6` which returns new sample lists (which are, at the virtual machine level, pointers to the initial, monaural sample list) for the left and right channels.

The code in Figure 6 represents physical units (such as the speed of sound in the air) and laws (e.g., the amount of space corresponding to every sample, depending on the sampling frequency) or parameters defining particular scenarios (such as the distance between ears).

We evaluated the different stages of optimization of the sound spatializer by processing a 120-second track while sampling the compass 10 times per second, using both the original version and an automatically specialized version (Section 3.4). Assessment is based on measuring the total processing time required and comparing it with the track duration, which indicates how well the bandwidth can be sustained by telling us how busy the processor is. We also recorded whether there were any artifacts such as clicks and silences. Their presence would reveal issues with garbage collection or swapping. The results are summarized in Table 1 where scenarios which generated acceptable sound are marked in boldface.

The code in Figures 5 and 6 can be compiled to bytecode and it can deliver spatialized sound with the required quality in a modern desktop or laptop computer, while responding in real time to the signals received from a compass. However it falls short in our target platform: generating stereo samples for a 120-second track takes 115.95 seconds, which means the processor is busy 96.6% ($= \frac{115.95}{120} \times 100$) of the time (Table 1). The remaining processor time is not enough to cope with the rest of the O.S. tasks without introducing noticeable clicks. To improve this situation we take advantage of the amenability of high-level languages to advanced program analysis and transformation in order to produce better executables without changing the original code. In

Compilation mode	Non-Specialized			Specialized		
	i686	Gumstix		i686	Gumstix	
	secs.	secs.	Utilization	secs.	secs.	Utilization
Bytecode	**4.70**	115.95	96.6%	**3.91**	103.49	86.2%
Compiling to C	**3.87**	98.08	81.7%	**3.36**	**88.27**	73.6%
Id. + semidet	**3.28**	**92.42**	77.0%	**2.85**	**83.74**	69.8%
Id. + mode/type annotation	**3.00**	**88.38**	73.6%	**2.57**	**79.42**	66.2%
Id. + arithmetic	**2.90**	**85.70**	71.4%	**2.47**	**78.01**	65.0%

Table 1: Speed results and processor utilization for a benchmark with different compilation regimes.

particular we used **(i)** partial evaluation (to specialize parts of the program), **(ii)** abstract-interpretation based compile-time analysis to ensure that the program will not raise any run-time exceptions (due to illegal modes, types, etc.) and to extract information in order to **(iii)** perform optimizing compilation to native code (via C) using the information on modes, types, determinism, and non-failure gathered during analysis.

3.3 Compile-Time Checking

The aim of compile-time checking is to guarantee statically that some program will satisfy certain *correctness criteria*, which in principle may be arbitrary. Static correctness proofs are certainly of utmost practical relevance in systems of high dependability or where updating the software is burdensome or costly. However, in most programming languages today the correctness criterion is type correctness, and compile-type checking boils down to type checking.

In the case of logic programs, arguments can in principle be input or output without further restrictions. This results in a very flexible programming language, where procedures are *reversible*. However, it is often the case that predefined (system) predicates require their arguments to satisfy certain calling conventions involving both types and modes (instantiation degree). Failing to satisfy such calling conventions is considered an error. For example, traditional Prolog systems check at run-time such calling conventions and errors are issued if the conventions are violated. In contrast to traditional CLP systems, in the Ciao analyzer and preprocessor, CiaoPP [10], information obtained by static analysis is used to reason about such calling conventions. To this end, the system has an assertion language [21] which allows explicitly and precisely stating calling conventions, i.e., preconditions for predicates. The Ciao system libraries are annotated to state pre- and post-conditions for library predicates. Several assertions expressing different pre-conditions and their associated post-conditions can co-exist for procedures which are multi-directional.

Static analysis in CiaoPP is based on abstract interpretation [7], and it is thus guaranteed to provide safe approximations of program behavior. Such safe approximations can be used in order to prove the absence of violations of a set of assertions, which can express more properties than just type coherence, and thus the absence of run-time errors.

For example, in the case of our implementation of the stream interpreter, we use the system predicate `is/2`. The arithmetic library in Ciao contains an assertion of the form:

```
:- trust pred is(X,Y) : arithexpression(Y) => num(X).
```

which requires the second argument to `is/2` to be an arithmetic expression (which is a regular type also defined in the arithmetic library) *containing no unbound variables*, and also provides the information that on success the first argument will be instantiated to a number. Analysis information using the *eterms* [27] abstract domain allows CiaoPP to guarantee at compile time that the program satisfies the calling conventions for system predicates (in this example just `is/2`) used in the program. Thus, the compiler *certifies* that no run-time errors will be produced during the execution of our code for the stream interpreter. The same applies to other predicates which access external entities (e.g., compass data) and whose behavior was modeled using Ciao assertions (see Section 4.2).

The user may optionally provide assertions for his/her own procedures. If available, CiaoPP will try to check at compile time such assertions. Clearly, the more effort the user puts into writing assertions, the more guarantees we have of the program being correct.

3.4 Partially Evaluating the Program

The code in Figure 6 performs repeatedly the same set of operations, many of them involving constants. While the part of the main loop dealing with arithmetic is not called a large number of times (because of the low sampling rate of the compass), opportunities for partial evaluation to improve execution time certainly exist. Indeed, all the code in Figure 6 is reduced to a single clause:

```
find_skip(A,B) :-
        C is sin(A*0.017453288889),
        B is round(25.94117647058824*C) .
```

Moreover, the calculations involving constant numerical values are performed at compile-time and the results propagated to the appropriate places in the program.[2] Loops and other parts of the program are also specialized, but the effect in those program points is less relevant. Input/output and other library built-ins are handled since they are appropriately annotated with assertions where they are defined.

Partial evaluation by itself gave, on average, speedups ranging from a factor of 1.15 to 1.2 on an i686 and around a factor of 1.1 on a Gumstix, when the compass is polled at 10Hz (see Table 1). On the Gumstix, partial evaluation decreases the processor utilization to 86.2% —substantially better than with the non-specialized code.

Although these results are encouraging, specialization by itself did not increase performance to a level where the spatializer really runs reliably in real-time on our target platform. Therefore, our next step towards gaining efficiency

[2]The reader may notice that C compilers also evaluate statically expressions containing constants. The situation is however different: in our case separate predicates (c.f., functions) are being evaluated statically guided by the calls made to them. If they were called from elsewhere in the program, the original definitions would have been kept together with the specialized versions.

(and, as before, keeping the initial code untouched) was to optimize away the bytecode interpretation overhead by compiling the Ciao program into native code, using progressively more compile-time information in order to generate code as optimal as possible.

4. TOWARDS OPTIMIZED NATIVE CODE

Two separate issues affect the performance of the sound spatializer: the time taken to process each sample, regardless of how it is processed, and the time taken to compute the new delay to be applied to the output streams. The former concerns mainly data-structure and control compilation (how the main loop is mapped into the lower-level language, how data structures are handled, and how data is read from and written to the streams). The latter is dominated fundamentally by costly (at least from the point of view of Ciao) floating-point arithmetic.

We attacked these problems by compiling to native code via C, using the schema presented in [16]. As we also wanted to identify the impact of different technologies in the efficiency of the application, we proceeded stepwise: we initially used only the information present explicitly in the original program, and later we used the extensive compile-time information gathered though global analysis.

4.1 Naive Compilation to Native Code

Compiling to native code without using information about types, modes, determinism, non-failure, etc. preserves exactly the data structures created when interpreting bytecode. Memory usage, existence (or not) of choice points, etc. do not change either, so any improvements in performance come mainly from reducing the time used in instruction fetching within the main virtual machine loop. Better data locality can help, but access patterns are difficult to predict and therefore this cannot usually be trusted as a source of improvement.

Despite the limited speedup that is obtained in the absence of additional information (Table 1), this was actually a turning point in our case: the processor utilization in the Gumstix decreased to 81.7% for the non-specialized program and to 73.6% for the partially evaluated version. The performance of the former is not enough to give a smooth playback; however, the latter is fast enough to play and to poll the compass at an adequate pace, while supporting some minimal additional load on the host processor. It is however not a satisfactory solution yet, as it was easy to produce noticeable interruptions in the playback just by adding a light load on the Gumstix.

4.2 Types, Modes, Determinism, Non-Failure

One of the tasks that non statically-typed languages have to perform at runtime is checking types and, for a logic-based language, also modes. Note that, unlike other declarative languages such as Mercury [23] or Haskell [11], Ciao programs do not need to include any type, mode, determinism, or non-failure declarations. Mode and determinism annotations are not needed in functional languages because all functions produce a single solution and their arguments are input.

Analysis information can be used to optimize native code generation in several points. For example, type information can be used to choose a more efficient, closer to the machine, representation. If mode information is also available, the

```
:- true pred new_sample_cycle(A,B,C,D,E,F,G,H)
    :  (int(A), term(B), int(C), term(D),
        term(E), term(F), rt2(G),  rt2(H))
    => (int(A),  int(B), int(C), int(D),
        rt2(E),  rt2(F), rt2(G), rt2(H))
    +  (is_det, mut_exclusive).

new_sample_cycle(0,4410,C,D,E,F,G,H) :-
        find_skip(~read_compass,D),
        skip(D-C,E,F,G,H).
new_sample_cycle(A,A-1,C,C,E,F,E,F) :- A > 0.

:- regtype rt2/1.

rt2([A|B]) :- term(A), term(B) .
```

Figure 7: Part of the information inferred for the compass program.

overhead involved in parameter passing and unification can be reduced by, e.g., compiling the latter into simple low-level assignments, perhaps with trailing. Last, determinism and non-failure information make it possible to reduce or avoid the creation of choicepoints since the compiler can know beforehand that no backtracking will be performed. This is, of course, only a partial list.

The analyzer we used (CiaoPP) is able to infer automatically a significant amount of information, provided that the *boundaries* of the program are well defined. For example, when there is communication with the outside world and the type of incoming data is relevant, then this data has to be described (via assertions in our framework). In our case study the only external data we need to deal with is that coming from the compass, since the sound samples themselves are treated as opaque data. Data coming from the compass is always a floating-point number. To reflect this, we added the following assertion for the `read_compass/1` predicate

```
:- trust pred read_compass(X) : var(X) => flt(X).
```

to the module encapsulating the compass access. This assertion should be read as: *"in any call to `read_compass/1`, the argument should be free when calling the predicate and it will be instantiated to a floating-point number upon success."* No other information is needed to infer accurate information regarding all the types, modes, and determinism of the whole program. However, if this information is not provided little useful information can be inferred and most of the improvements that will be described in the following sections cannot be achieved. We want to note that in bigger, modular applications, boundary information is usually provided as part of the module interfaces (and it may have been automatically inferred), or it can be generated if all source code, libraries included, is available.

Figure 7 shows a selection of the information CiaoPP can deduce for the predicate `new_sample_cycle/8`. Much more information on sharing (pointer aliasing) and freeness (pointer initialization) was produced, which we omit since it is not instrumental for our case. However, it would be vital if we were to parallelize the code automatically.

`read_compass/1`, as we discussed previously, performs communication with the concurrent process that reads the compass, and its behavior is modeled with the assertion previously shown. With this information, the predicate `new_sample_cycle/8` is inferred to be deterministic and the clauses are found to be mutually exclusive (as expressed by the

(is_det, mut_exclusive) assertion). This means that a more efficient compilation scheme, which does not produce superfluous code to handle backtracking, can be used.

Additionally, the open-ended list used to hold the samples to output is approximated with the type rt2/1, which only states that the argument is a *cons* cell. This information, albeit not complete, is enough for a lower-level compiler to generate better code which avoids testing at runtime the type of a parameter.

If determinism and non-failure inference are used, the processor utilization is reduced to 77% (for the non-specialized program, which is now able to generate stereo samples and poll the compass simultaneously with quite acceptable sound) and to 69.8% (for the specialized version). If mode (variable instantiation state at predicate entry and exit) and type inference are also used, the processor utilization gets further reduced to 73.6% and 66.2% for the non-specialized and specialized programs, respectively.

4.3 Optimizing Arithmetic Operations

The strategy for compilation to native code used so far preserves the original data representation of the WAM: data is still stored in tagged words (i.e., *boxed*). This does not incur a big performance penalty in most cases, since C compilers generate efficient code to do the tagging/untagging, and the overhead is, in general, relatively small in comparison with what is done with the data itself.

This overhead is however comparatively large for operations which are simple enough to be translated to a single assembler instruction. Arithmetic operations stand out, and floating-point arithmetic suffers from an additional overhead: floating-point numbers are not carried around directly in a tagged word; rather, the tagged word points to a structure which holds the floating-point number. Therefore, boxing and unboxing a floating-point number are comparatively costly operations which, in principle, have to be repeated every time a floating-point operation is performed. Additionally, keeping floating-point numbers boxed needs more memory and garbage collection has to be called more often.

Another disadvantage of keeping numerical values in boxed form is that when compiling to native code via C, the C compiler does not *see* native machine data (e.g., ints, floats, doubles), since they are encoded inside tagged words. This makes it difficult for the compiler to apply many useful optimizations (instruction reordering, use of machine registers, inlining, etc.) devised for more idiomatic C programs.

Unboxing has been studied and applied in functional programming [13, 18] with good speedup results. This is helped in part by the use of strict type systems and the lack of different instantiation modes. Strict typing (and compulsory information about modes and determinism) applies also to the case for Mercury, which does not need boxing and unboxing. An interesting related approach is that of [20] for Haskell, where the kernel language was augmented with types to denote explicitly unboxed values and the simplifications to remove redundant operations were formalized as program transformations. However, language differences and the issues that that work focuses on (strictness, polymorphism, etc.) makes applying directly these techniques difficult in our case.

Unboxing for CLP systems, which are untyped and dynamically tagged, has received comparatively little attention. For example, Aquarius [26] did not perform boxing/unboxing, and mainstream CLP systems, such as SICStus, do not use it when compiling to native code. The closest work is perhaps [1] which proposes a compilation strategy for the concurrent, committed-choice logic language Janus which, starting from a program annotated with type and mode declarations, performs a series of analysis to determine the best representation of each procedure argument and to avoid redundant boxing/unboxing operations.

We share in fact some ideas with [1], although the languages are quite different. Similar type and mode annotations are required, which are inferred automatically in our case. However, we have to infer also information about determinism and non-failure, which is implicit in the language design of Janus, as it does not support backtracking or failure. A similarity with [20] is that we have formulated the solution as a source-to-source transformation on an extended language which includes boxing and unboxing operations. The implementation used in our experiments supports unboxed representations for some basic, native types and for temporal variables with a restricted lifetime, in order to ensure that there will be no interaction with garbage collection.

Our approach to boxing/unboxing removal works by exposing the code of builtins which inspect word tags. They typically share a similar structure: perform type checking on input arguments, unbox values, operate on them, box output values, and unify with output arguments. Informally, the process we use to detect and remove unneeded boxing/unboxing changes is:

1. Unfold builtin definitions to make type checking, unboxing, and boxing visible.

2. Make a forward pass to remove redundant unboxing operations. An abstract state relating boxed variables with their unboxed version is kept. It is updated with each unbox operation by adding a pair of linked variables (corresponding to boxed/unboxed views of the same entity) and by removing the pair when the versions become out-of-sync or, for temporal variables, when they become out of scope. This state is consulted to check for the availability of unboxed versions of variables when needed.

3. Make a backward pass to remove unnecessary box operations whose result is not used any longer.

Figure 8 sketches how the algorithm behaves for a short piece of code corresponding to the body of find_skip/2 (Section 3.4). The initial code is shown in the box at the top left. The next box contains the same code after splitting arithmetic expressions into basic operations which still work on boxed data and adding number-creation primitives. Each of these primitives is later expanded into smaller components which either create or disassemble boxed values or work directly with unboxed numbers.

Dashed lines with arrows relate pairs of unbox / box operations which can be simplified by a forward pass, since the boxed versions of the variables are not used between them. Goals marked with [c] denote checks (coming from builtin expansion) which are statically known to be true at runtime, either because of assertions at source level or thanks to information gathered in the fragment of code being compiled. They can be safely removed. Finally, goals marked with [u] are marked as unnecessary during the backward pass because their output value is not used.

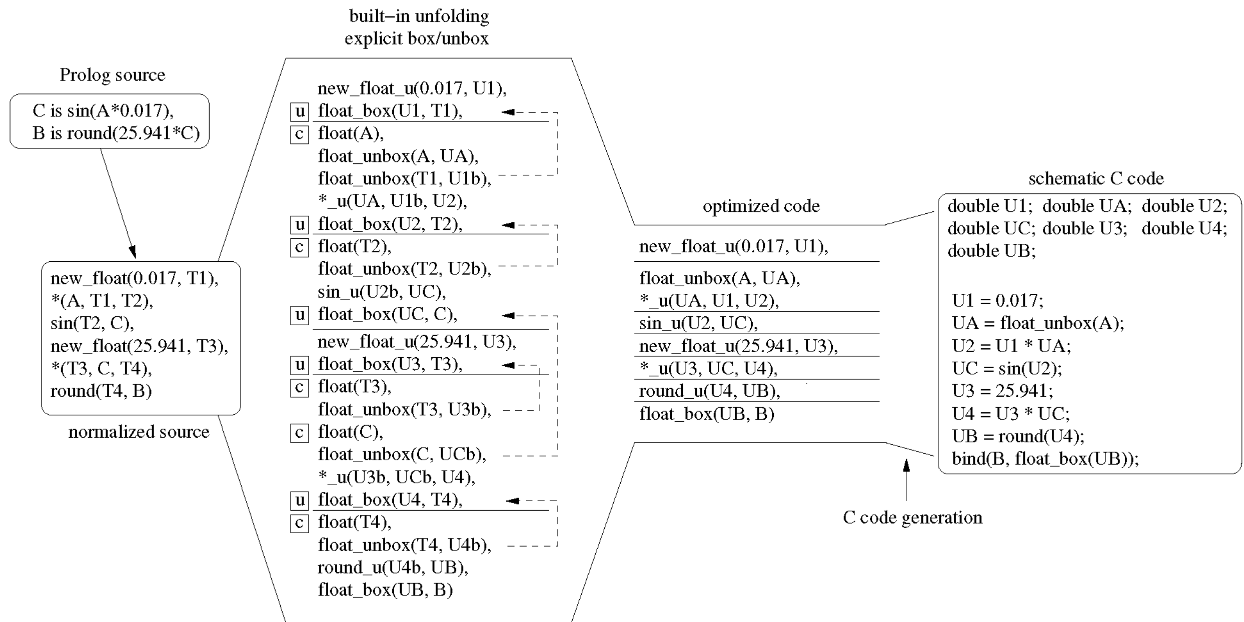

Figure 8: Unboxing optimization.

The next two stages show the intermediate program after removal of dead code and, finally, the corresponding C code. Only one boxing and one unboxing operations (for the input and output parameters, respectively) are needed, and intermediate variables have been mapped to C (native) variables. Additionally, since mode information tells us that the second argument is always free variable, only a very specialized form of unification (the call to bind(), in fact a pointer assignment with trailing) is needed.

As before, this optimization was applied both to the non-specialized and to the specialized Ciao program, leading to some performance gains: the unboxing optimization made it possible to reduce processor utilization to 71.4% for the non-specialized program and to 65% when running the specialized one. In both cases this is enough for the Gumstix to respond adequately to compass movements, even if there are several other (non CPU-bound) processes running on it.

5. SUMMARY OF THE EXPERIMENTS

Although we already presented some results in the previous sections, we will summarize our experiments and put them in the light of a new scenario we did not discuss before in order to make the presentation as clear as possible. A rough classification of the experiments performed, the processor utilization, and a pictorial summary of their characteristics, is shown in Figure 9.

5.1 Basic Results

All tests were run on a Gumstix, as commented throughout the paper, and on a SpeedStep Centrino @ 1.4GHz. Table 1 shows performance figures for both. While the input stream is not infinite, after a few seconds both CPU usage and memory consumption stabilize, which makes us confident that the program would be able to run indefinitely.

The original non-specialized program running on a virtual machine is fairly efficient, especially taking into account that it is written in a style which is very close to a specification: buffer sizes are not stated anywhere (they self-adjust

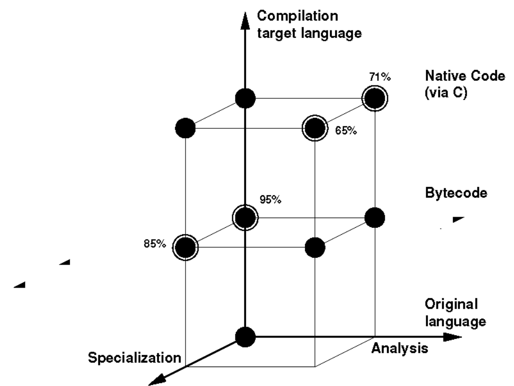

Figure 9: Global view of the experiments.

dynamically), memory management is automatic, etc. But there is not enough spare time to produce a sustained high quality sound stream on a Gumstix. A combination of specialization plus compilation to C, or compilation to C plus compile-time information, is enough to make the program deliver acceptable sound. However, the CPU usage in the Gumstix is still too high and any other activity on the same board causes audible interferences. It is only when both specialization plus analysis information are used to compile to C that other processes can be supported on the same board without noticeable interferences.

The best version runs, on the Gumstix, 1.5 times faster than the initial one. The difference is larger for the i686 case, as the speedup is around 1.9. However, those speedups also depend on particular scenario characteristics, such as polling frequencies, and, as we will see, other scenarios can exhibit very different behaviors.

5.2 Increasing the Sampling Frequency

The optimizations on arithmetic operations affect mainly a tiny fragment of code which computes the phase shift be-

Compilation mode	Non-Spec.	Specialized
Bytecode	25.64	14.00
Compiling to C	21.59	11.99
Id. + semidet	19.59	11.53
Id. + modes/types	19.19	11.08
Id. + arithmetic	6.97	3.62

Table 2: Results with a higher compass polling rate.

tween the two ears and which is executed infrequently (10 times per second) with the current compass hardware. A faster poll rate, or the need to process other signals coming at a higher frequency would require a larger fraction of processing time to be spent on computing the heading data.

To set up an extreme situation, we have simulated the case where heading data is provided at the same rate as the audio data (44,100 Hz). Note that this is the highest polling rate which makes sense, since a faster rate would actually discard compass data until the next audio sample is available. Table 2 summarizes the results under that assumption for an i686. In that scenario we measured a 7-fold speedup between the slowest and the fastest executable. This is indeed a very good result, and an extrapolation to the Gumstix suggests that with our current analysis and compilation technology the software running on the Gumstix would be very close to supporting compass sampling at 22,050 Hz.

The improvement introduced by using unboxed data and by specializing the program is much higher than in the previous set of tests. The reason is the same for both cases: more time is comparatively spent on arithmetic operations. Therefore, compile-time specialization, which evaluates many floating-point operations at compile time, simplifies fragments of code whose execution would take a substantial portion of the execution time (compare the left and right columns in Table 2). Something similar happens with the low-level optimization of floating-point arithmetic: operations are not removed, but they become much cheaper instead (last and next-to-last rows in Table 2)

5.3 A Comparison with C

We wanted to determine how far we are from an implementation written directly in C. We wrote a C program which mimics the Ciao one in the sense that it offers the same flexibility: it uses dynamic memory, buffer size is not statically determined, etc. It was written by an experienced C programmer and it does not incur any unnecessary overheads. The results are highly encouraging: the C program was only between 20% (for the tests in Table 2) to 40% faster (for the tests in Table 1) on an i686 processor. Interestingly, this C program did not behave as smoothly as expected when executed on the Gumstix: memory management caused audible clicks, and writing an *ad-hoc* memory manager would probably have been needed — or sacrificing flexibility by using static data structures. Additionally, the complexity of the C code would have made tuning the application much more difficult.

6. CONCLUSIONS AND FURTHER WORK

In this paper we have shown how a set of advanced analysis, transformation, and compilation tools can be applied to a program written in a high-level CLP language which deals with a combination of numerical and symbolic processing (in the form of data structures) to generate an executable which runs adequately in terms of time, memory, and feedback to the user on a pervasive computing platform. We believe that the techniques we show here can be effectively used in a broader set of scenarios.

The application we used is a sound spatializer, intended to run on a wearable computer, the Bristol "CyberJacket". There were hard requirements regarding timing, sound quality, and non-functional behavior. The application code was deliberately not "tricky", but clear and as declarative as possible; it was not changed or adapted (by hand) in any of the experiments. The initial executions (using a bytecode interpreter in the wearable computer) did not meet the stated requirements, but a series of analysis, specialization, and optimizing compilation stages, which we reported on, managed to make it run well within spec on the target machine. All of them were carried on using the Ciao/CiaoPP programming environment. In an alternative, more demanding scenario, needing more arithmetic operations, our code performs within 20%-40% of a comparable C program.

It is difficult to single out a compilation stage which can be attributed the majority of the benefits. In the first (non arithmetic intensive) scenario, specialization caused most of the speedup because of the reduction in the number of arithmetic operations and calls performed. However, in the second scenario, boxing / unboxing removal was the clear winner. The rest of the optimizations were not highly relevant in this case, but we believe they would have been if more symbolic processing were needed. In any case, the information gathered by the analysis was also used by the low-level optimizing compiler.

We intend to continue the development and integration of advanced compilation techniques. In particular, we want to address inter-procedural and inter-modular boxing and unboxing, as well as to explore the tradeoffs of doing additional boxing/unboxing steps, which has an extra overhead but which in general may benefit other parts of the code, and generating automatically "hints" for the garbage collector. We also want to study compilation schemes aimed at saving memory space which, although not a problem in our case study, can be a concern in other scenarios.

8. REFERENCES

[1] P. A. Bigot and S. K. Debray. A Simple Approach to Supporting Untagged Objects in Dynamically Typed Languages. In *International Logic Programming Symposium*, pages 257–271, 1995.

[2] J. Blauert. *Spatial Hearing : The Psychophysics of Human Sound Localization.* The MIT Press, 1983.

[3] G. Bolella and J. Gosling. The Real-Time Specification for Java. *IEEE Computer*, 33(6), June 2000.

[4] F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. P. (Eds.). The Ciao System. Reference Manual (v1.10). Technical report, School of Computer Science (UPM), 2004. Available at `http://clip.dia.fi.upm.es/Software/Ciao/`.

[5] M. Carro and M. Hermenegildo. Concurrency in Prolog Using Threads and a Shared Database. In *1999 International Conference on Logic Programming*, pages 320–334. MIT Press, Cambridge, MA, USA, November 1999.

[6] A. Casas, D. Cabeza, and M. Hermenegildo. A Syntactic Approach to Combining Functional Notation, Lazy Evaluation and Higher-Order in LP Systems. In *Eighth International Symposium on Functional and Logic Programming (FLOPS'06)*, Fuji Susono (Japan), April 2006.

[7] P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of POPL'77*, pages 238–252, 1977.

[8] G. Gupta, E. Pontelli, K. Ali, M. Carlsson, and M. Hermenegildo. Parallel Execution of Prolog Programs: a Survey. *ACM Transactions on Programming Languages and Systems*, 23(4):472–602, July 2001.

[9] M. Hermenegildo, F. Bueno, D. Cabeza, M. Carro, M. García de la Banda, P. López-García, and G. Puebla. The CIAO Multi-Dialect Compiler and System: An Experimentation Workbench for Future (C)LP Systems. In *Parallelism and Implementation of Logic and Constraint Logic Programming*, pages 65–85. Nova Science, Commack, NY, USA, April 1999.

[10] M. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming*, 58(1–2):115–140, October 2005.

[11] S. P. Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

[12] E. A. Lee. Overview of the Ptolemy Project. Technical Report UCB/ERL M03/25, University of California at Berkeley, July 2003.

[13] X. Leroy. Unboxed Objects and Polymorphic Typing. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 177–188, Albequerque, New Mexico, 1992.

[14] M. Leuschel. Design and Implementation of the High-Level Specification Language CSP(LP). In I. V. Ramakrishnan, editor, *PADL'01*, volume 1990 of *Lecture Notes in Computer Science*, page 14. Springer-Verlag, March 2001.

[15] M. McCarthy and H. Muller. No Pingers: Ultrasonic Indoor Location Sensing without RF Synchonisation. Technical Report 003-004, University of Bristol, Department of Computer Science, May 2003.

[16] J. Morales, M. Carro, and M. Hermenegildo. Improving the Compilation of Prolog to C Using Moded Types and Determinism Information. In *Intnl. Symposium on Practical Aspects of Declarative Languages*, number 3057 in LNCS, pages 86–103. Springer, June 2004.

[17] H. Muller and C. Randell. An Event-Driven Sensor Architecture for Low Power Wearables. In *ICSE 2000, Workshop on Software Engineering for Wearable and Pervasive Computing*, pages 39–41. ACM/IEEE, June 2000.

[18] J. Peterson. Untagged Data in Tagged Environments: Choosing Optimal Representations at Compile Time. In *Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 89–99. ACM Press, September 1989.

[19] J. Peterson, P. Hudak, and C. Elliott. Lambda in Motion: Controlling Robots with Haskell. In *PADL*, pages 91–105, 1999.

[20] S. L. Peyton Jones and J. Launchbury. Unboxed Values as First Class Citizens in a Non-strict Functional Language. In J. Hughes, editor, *Proceedings of the Conference on Functional Programming and Computer Architecture*, pages 636–666, Cambridge, Massachussets, USA, 26–28 August 1991. Springer-Verlag LNCS523.

[21] G. Puebla, F. Bueno, and M. Hermenegildo. An Assertion Language for Constraint Logic Programs. In *Analysis and Visualization Tools for Constraint Programming*, pages 23–61. Springer LNCS 1870, 2000.

[22] C. Randell and H. L. Muller. The Well Mannered Wearable Computer. *Personal and Ubiquitous Computing*, 6(1):31–36, February 2002.

[23] Z. Somogyi, F. Henderson, and T. Conway. The Execution Algorithm of Mercury: an Efficient Purely Declarative Logic Programming Language. *JLP*, 29(1–3), October 1996.

[24] R. Stevens. A Survey of Stream Processing. *Acta Informatica*, 34:491–541, 1997.

[25] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A Language for Streaming Applications. In *International Conference on Compiler Construction*, number 2304 in LNCS, pages 179–196. Springer Verlag, 2002.

[26] P. Van Roy. *Can Logic Programming Execute as Fast as Imperative Programming?* PhD thesis, Univ. of California Berkeley, 1990. Report No. UCB/CSD 90/600.

[27] C. Vaucheret and F. Bueno. More Precise yet Efficient Type Inference for Logic Programs. In *Proc. of SAS'02*, pages 102–116. Springer LNCS 2477, 2002.

[28] M. Wallace. *Functional Programming and Embedded Systems*. PhD thesis, York University, January 1995.

[29] G. Welch and G. Bishop. An Introduction to the Kalman Filter. Technical Report TR95-041, Department of Computer Science, University of North Carolina - Chapel Hill, November 1995.

[30] M. Wolfe. How Compilers and Tools Differ for Embedded Systems. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*. ACM and IEEE Computer Society, September 2005. Keynote Speech.