ASAP

IST-2001-38059

Advanced Analysis and Specialization for Pervasive Systems

Requirements Report

Deliverable number:	D3
Workpackage:	Requirements and Case studies (WP2)
Preparation date:	1 August 2003
Due date:	1 May 2003
Classification:	Public
Lead participant:	Univ. of Bristol
Partners contributed:	Univ. of Bristol, Univ. of Southampton, Roskilde Univ

Project funded by the European Community under the "Information Society Technologies" (IST) Programme (1998–2002).

Abstract

The objective of work-package 2 is to find a suitable set of case studies to be used in ASAP. The case studies will be selected from existing pervasive systems, or else chosen to investigate new pervasive applications that become feasible when powerful development tools are available. This document describes the issues that are important in the selection of case studies. We first present the field of pervasive computing and the metrics that can be applied to describe the quality of pervasive systems. For example, one version of a system may be better than another because it runs faster or uses less memory. Some of the metrics are quantitative (seconds, milliwatts), whilst others are of a more qualitative nature (expressiveness of a programming language, or usability of tools). We can use these metrics to evaluate the effectiveness of specialization and analysis tools. After that we discuss the area of program specialization and analysis, and in particular the advances that are relevant in a pervasive context. We conclude the deliverable by outlining two example case studies and summarizing the requirements for case studies.

Contents

1 Introduction

The aim of the ASAP project is to study the application of specialisation and analysis techniques to the domain of pervasive computing. The goal is to develop a novel theory of specialisation which takes into account the specific requirements of the pervasive domain. In this document we will examine these requirements and how they will determine the case studies used on the project. In order for the results of our experiments to be relevant to our project domain and applicable in the real world we must define the correct criteria for our systems to be measured against.

To make sure that the case studies are of a realistic nature it is beneficial to use problems taken from an existing pervasive system. We should however not limit ourselves to existing pervasive applications as without the novel techniques that we are studying they will not fully utilise the available resources. Some applications may not have been feasible without this advancement in technique and so were not developed at all. We will identify the restrictions on resources that are specific to the pervasive domain.

In this document we present the requirements for the case studies. The case studies themselves are going to be presented in a separate deliverable (D7-2.2, Case Studies). We will first present the background on wearable and pervasive computing in Section 2. After that we will present the metrics to evaluate software that is used in a wearable and pervasive context in Section 3. In Section 4 we discuss the challenges that stem from the restrictions imposed by program transformations. These restrictions limit what can realistically be analysed automatically, and we must keep these restrictions in mind when selecting case studies. We will then present an example case study in Section 5. We conclude with a summary of the requirements analysis.

2 Background on Wearable and Pervasive Computing

In this section we present the life-cycle of pervasive systems. We start by clarifying what we mean with a pervasive system, give some examples, and then discuss design, implementation and maintenance aspects.

2.1 Embedded, Wearable and Pervasive Systems

It is difficult to precisely define pervasive systems. Below we define pervasive systems in relation to embedded systems and wearable systems. The simplest view is that the class of wearable systems is a subset of the class of pervasive systems, which in turn is a subset of the class of embedded systems, as shown in Figure 1.

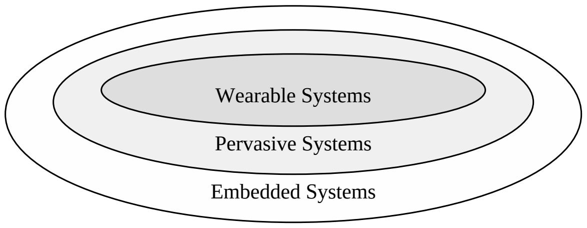

Figure 1: Relation between embedded, pervasive and wearable systems

It is possible to describe a pervasive system in two ways, the first is the less technical description and defines a pervasive system in terms of its interaction with users and its physical embedding within the environment. The second description defines a pervasive system in relation to embedded systems by way of highlighting the differences between the two. There is no exact definition of pervasive computing and so it is not possible to provide exact boundaries of what is and what is not a pervasive system. We will try and use a definition that corresponds closely to other researchers in the field, but this is a grey area and there will be some overlap and some differences.

The less technical definition of a pervasive system is in terms of what it is not. A traditional computer system (such as a desktop computer) is a general purpose machine that can be configured by the user to perform many different tasks. It uses very general methods of communication with the user, a keyboard and a mouse allow a wide range of modes of interaction. User interfaces can give a variety of visual and auditory cues to the user. Furthermore, we can assume that the user is not engaged in any other tasks, that they can give the interface to the computer their full attention and that there are no barriers to the interface.

A pervasive system is a collection of specialised devices, while the overall system can be reconfigured for different purposes each individual device will have a fixed function and reconfiguration will be through changing how devices interact with each other. The goal of a pervasive system is that computing has pervaded the environment, this means that we have augmented the utility of the environment. The focus of the system is no longer on interacting with a computer and there may be barriers to traditional methods of input. A keyboard or a mouse is very convenient and easy whilst sitting at a desk but become cumbersome and awkward whilst walking down a street. The barriers that stop the use of traditional input and output devices are not just the physical placement of devices, there are also psychological barriers; Interaction has to be integrated with other activities that the user may be engaged in. Interaction needs to become unconscious and indirect.

A more technical definition of a pervasive system contrasts it to an embedded system. An embedded device is one in which a computer is integrated (or embedded) into a physical device.

The usual purpose of this is to allow the computer to monitor the devices' operation and to react to changes by controlling the device. The typical usage is to keep the device operating within predefined parameters. There are many examples as the number of installed embedded devices is much greater than the number of traditional computer systems. Cars and aircraft both have many individual embedded systems that look after specific aspects of their operation. A car will contain embedded devices that included engine management systems that control the timing of combustion and the flow of fuel and exhaust gases (the S-class Mercedes Benz has 65 embedded processors [27]). These allow improved performance over their purely mechanical rivals.

Pervasive systems are a subset of embedded devices. There is a lot of overlap between the two and large areas of common theory from embedded devices that can be exploited for use in the design of pervasive systems. One characteristic that differentiates between the two is that pervasive systems tend to comprise a relatively large number of devices that interact with each other. Embedded systems tend to be constructed from a smaller number of devices that interact with the user or the hardware being automated.

Wearable systems are a further subset of pervasive systems. These are a specific type of pervasive system that has been integrated with the users' clothing, e.g. a jacket. These are highly mobile systems as they are mounted on a person rather than being a part of the environment around them. They can measure aspects of the users' context such as his position and movement. This information can then be used to augment the user's experience, allowing interaction with virtual objects. Feedback from the wearable can be visual (using a headset) but this is often too intrusive. Audio feedback is more peripheral allowing the user to carry on with other activities while using the jacket. The communication abilities of the jacket open up new applications that can augment social contact, providing background information about people or helping to form social networks with common needs or goals.

These descriptions are not exact, there are no firm boundaries between the different classes of device and some devices may belong in several categories. Consider the case of the PDA. This does not fit the traditional model of an embedded device. Neither does it seem to be a particularly pervasive device, but some people do regard it as a wearable device - even though it lacks integration with the users clothing and does not have access to the contextual data that we associate with wearable computing.

In this document we will talk about *systems* and *devices*. A device is one of the components of a system. As an example, a user may carry one or more devices (for example a PDA and a mobile phone), which communicate via blue-tooth, and use them to control the central heating at home (press a button on your PDA, the phone uses GPRS to connect to a server at home connected to the central heating controller). The combination of PDA, Mobile Phone and the server is a pervasive system. The PDA is a wearable device.

2.2 Example Wearable and Pervasive Systems

The design of pervasive and wearable systems is not a new field. The first smart homes were designed decades ago, but it is only with the advent of cheap, low power, highly integrated computing devices that it has become feasible to see these as commodity items rather than the plaything of rich industrialists [14].

The earliest designs [12] for pervasive systems created devices with entirely new functionality, as the cost of components has reduced over the years some new devices have appeared that replicate the function of non-computer based systems (such as digital cameras). As the size and cost of these devices decreases they are increasingly being integrated together causing a convergence between many different types of device. This can be seen in the smart-phone market where the functionality of digital cameras is being integrated with mobile phone technology and the functionality of a PDA.

2.2.1 New services

New services are devices that offer something for which no automated solution was available. The examples we present below are smart-cards and GPS receivers.

The smallest device that people carry around is the smart-card. Smart-cards are used in satellite TV, digital cable, credit cards and mobile phones. Every smart card has a tiny embedded computer system, measuring 5x5 mm. Unlike other systems it does not carry its own power, but it is only powered up when the user inserts their smart-card in a terminal, such as a phone, or ATM. The most important property of a smart-card is that it is tamper-proof. The owner of a smart-card cannot get into it, hence the issuer of a smart-card can store some secret information on it which the owner can use to identify themselves. This way, the calls made from a mobile phone are billed to the right person, and when a credit card is used the right account is debited. Smart-cards in this form are a digital version of an identity card. When the processor on a smart card is made more powerful, the smart card can be used for other purposes, for example it can store (digital) money.

A bigger wearable device that is now common use is the GPS receiver. GPS (Global Positioning System) was developed by the military, but is now used in yachts, cars, and in hand held devices used by hikers and climbers. GPS works by measuring the relative distance to at least four satellites orbiting earth. Using these distances, the module can calculate its latitude, longitude and altitude. However, to get as high as 5 meter accuracy, arrival times will have to be measured with high accuracy (~ 10ns), and a filter will have to be applied that eliminates noise and that estimates the user's location. We will come back to this in Section 5.

2.2.2 Existing services

The two devices we present below, digital cameras and mobile phones, are replacements for the analogue cameras and telephones that have been around for a century. Both of those devices originally just provided the same services as their analogue counterpart. But in both cases, extra services have been provided since.

Digital cameras are now serious competitors to film-based cameras. In their simplest form they just store pictures digitally rather than on film. But more complex cameras can store the camera's orientation (to recognise portrait pictures), aperture, focal length, exposure time, etc. Indeed, when integrated with a GPS device, the photograph can be augmented with the location, allowing the development of subject based albums.

A large fraction of the population carries a mobile phone. The first generation of phones only provided the interface of a phone. Later generations added information retrieval services such as WAP and HTML over GPRS. The latest generation also provides location based services. These phones can tell you where the nearest pizza restaurant is, or order a taxi to where you are.

2.2.3 Augmented every day devices

The devices presented in the previous two sections are devices that we use explicitly. Another use of pervasive systems is to embed them in our everyday environment.

One prime place to embed a pervasive system is in toilets. Toilet lights that switch on when people walk in, toilets that flush themselves when you get off the seat, and taps that open when you put your hand under them are all well established. The prime reason for the development of those systems is hygiene (since people do not have to touch a light switch, tap etc), but since an automatic light switch is convenient and saves power, they have also been introduced in offices. At present these systems use minimal processing power, and have zero intelligence.

Recently, Danish researchers have developed a DIY flat-pack furniture with an embedded computer system. The sensors embedded in the flat-pack tell the person assembling the flat-pack whether the assembly is right or not. It will check for orientation, and whether pieces are in the right place relative to each other [2].

Benetton (the fashion house) and Philips Semiconductors have toyed with the idea of using smart chips in clothing labels. These chips would replace a bar-code, and allow Benetton to track their clothes from factory to shop. Benetton has not yet decided whether to go ahead with it, because of fears that people can be tracked [24]. Tagging clothes labels has some great potential — it enables a washing machine to find out what it washes and select an appropriate programme.

Christmas cards have long been safe from innovation. But over the last five years silicon has become so cheap and small, that cards are on the market that have music recorded on them. When the card is opened, a switch is closed and the music is played. This device is not pervasive or wearable, but one of a next generation, called *disposable devices*.

2.2.4 Future devices

Recently, we have seen the production of Java enabled phones, many blue-tooth enabled devices, and it will not be long before applications are developed that rely on information from various devices.

2.3 Implementation aspects of wearable systems

In the development of a pervasive system, the designer is constrained mostly by the physical size of the device and its power requirements. As such, they will choose a device that is generic enough for implementing the required functionality, but small enough that it consumes very little space. Usually there are various independent parts in a pervasive system, each of those parts has its own constraints on size and power issues, but there is only one global specification as to what the system should do.

As an example, we present a number of ways to implement ultrasonic location systems. Ultrasonic location systems allow a wearable to establish its location. These systems work by measuring distances between the wearable and known places in the infrastructure, and then performing a tri-lateration on those distances. The distances are measured by sending an ultrasonic "chirp", and measuring its time of flight. By multiplying the time of flight with the speed of sound, we can establish the distance, and with three distances between known points and the wearer, we can establish where the wearer is. The difficult bit is that the transducer sending the ultrasonic chirp must agree with the receiver when the signal was sent, otherwise we do not know the time of flight. Below are some solutions to this:

- Equip the wearer with a radio receiver and ultrasonics transducer, and mount three or more receivers in the room. On a radio signal transmitted by the computer in the room, the wearable will send out a chirp, and we measure the time of arrival in each of the receivers. By subtracting the time that we sent the radio signal, we can calculate time of flight, and calculate the wearer's position. Note that the infrastructure now knows where the wearable is: if the wearable is to know, the location information must be transmitted to the wearable. [21]

- Equip the wearer with a radio receiver and ultrasonic receiver, and mount three or more transducers in the room. The computer in the room will transmit a radio signal and chirp for each transmitter. The wearable receives the radio signal and chirps, calculates the time difference, and obtains the times of flight. The wearable can now calculate its location, if the computer in the room is to know, the information will have to be transmitted from the wearable. [28]

The advantage of the first solution is that the part of the system on the wearer consists of very few components: a radio receiver and an ultrasonic transducer. The computational work (trilateration) is performed on computer infrastructure, which can be mains powered. The advantage of the second solution is privacy. The location is computed on the wearable, hence the wearable knows its location (rather than the infrastructure knowing the wearer's location). By placing the transducers in a square, the computation can be reduced to squaring four numbers, an operation that even a micro-controller can do. So, depending on how the system is set up, we need to do computations on one side or the other. In bigger systems, the implementation can usually be split over many devices.

A second issue is the one of hardware/software trade-off. There are various types of hardware that can be used to implement pervasive systems. Some of the tasks traditionally required hardware, but can now be implemented in software, thanks to the increase in processor speed. So instead of needing banks of filters in hardware, one can filter in software. People have successfully tried to create standard architectures for pervasive systems, often composed of a low power processor (for example a MicroChip PIC [18], or Intel StrongARM [13]), some sensor/actuators, and some mobile communications interface.

Not all pervasive systems have a software component. There are parts of pervasive systems that can be completely hard-coded. As an example, the active clothing labels described in Section 2.2.3 are devices without software (and even without battery!). The tags consist of an aerial that can capture sufficient energy to transmit an RF-ID. The ID is hard-coded in the tag.

2.4 Mobile Software

There is a classic trade-off between the time taken to specialise a piece of software and the potential speed increase from specialisation. An analogous trade-off exists when using specialisation to reduce power usage. Any specialisation process will itself consume energy during execution. The desired result of a specialisation is a reduction in the energy consumed by the specialised program during execution. If the energy saved by the execution(s) of the target program is not greater than the energy consumed during specialisation then overall the energy efficiency will be decreased. This will result in an increase in the device's power usage.

The combination of mobile devices and pervasive wireless networking changes the impact of specialisation on energy usage. We can offload processing from a mobile client device to a server that has no power restrictions. One method of performing this action is the use of mobile software; processes that can migrate from one node to another in a network during execution. This method reduces the energy equilibrium point necessary to make specialisation viable and could provide a simpler starting point for research.

One of the aims of mobile software can be to allow processes to migrate to idle units. Several low speed processors will consume less power than a single high speed processor by virtue of the fact that clock speed is proportional to power squared. Utilising previous research on strip-mining parallelism and distributing it through mobile processes will yield a further reduction in power consumption. We are not going to pursue this in ASAP.

2.5 Pervasive Software Life-Cycle

In ASAP we are initially looking into systems that do have a software component. The interest of ASAP is to improve this software component, for example reduce its execution time. Section 3 contains details on what we expect to improve. In order to understand why the metrics are important, we first describe the life-cycle of software in a pervasive system.

Because of the nature of a pervasive system, software is usually developed using a host-target approach. This means that software is developed on a host-computer, cross-compiled to the target platform, loaded onto the target platform and executed. Depending on the system, the target platform may run some form of an operating system (for example embedded Linux or Windows-CE), or it may just be raw hardware (for example a micro-controller). The software life-cycle is sketched in Figure 2.

The traditional "implementation" box has been expanded in this figure, to detail the process that happens inside it. There are three important issues that are particular to pervasive computing:

- The execution of a pervasive system is in many cases battery powered, which means that only a limited number of instructions can be executed. For example, on a watch battery (1.5V 18 mah), one can execute around 10 billion instructions (we will come back to this in the next section). If the program is made twice as efficient in terms of the number of instructions executed, then the device can, in theory, run twice as long on the battery.

- The person coding the software will spend a long time actually optimising the code so that it will work under the constraints presented. A PIC has a program memory that can store in the order of 1000 instructions, and has 100 bytes of memory for holding its data.

- Cross-compilation and loading are two explicit steps. Depending on the nature of the system, the software is loaded explicitly via some link (for example a cradle or a wireless

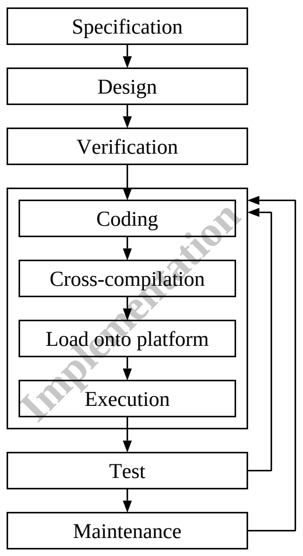

Figure 2: Pervasive software life-cycle

network), or implicitly on demand (for example Java classes). Program optimisation can take place on either side of the loading phase: during the compilation of the code, or when loaded on the platform. The latter type of optimisation might be beneficial given that programs are generic, but part of their input depends on the place they are in.

3 Metrics for evaluating Pervasive Software

In this section we discuss the metrics that we are going to use to identify whether we successfully improve pervasive software. We derive those metrics from the requirements of pervasive devices. Ideally, pervasive devices have:

- Low power consumption
- High performance
- Small footprint (small physical size)

The issue of low power consumption is especially relevant for mobile devices. In a mobile system the power is supplied by a battery. The battery weight varies linearly with the amount of energy that we can store. In a non mobile system, low power is less important if systems are mains powered, but still desirable as a low power system does not need a fan and is hence quiet.

We have termed the list above an ideal set of requirements, because low power consumption forms a trade-off with high performance. The high-performance in a small-footprint trade-off is common to many systems, but achieving high performance in a low power system is specific to the design of mobile devices. We are attempting to maximise both power efficiency and performance in our system. This implies that we cannot rely on continuous advances in hardware performance to deliver improvements in software quality. Instead we must increase the efficiency of the software.

3.1 Power efficiency

The power efficiency of a system is the inverse of power consumption. To reduce power consumption we must first reduce the number of instructions that the processor executes. Once the number of instructions has been minimised, we can reduce the power consumption of the system in a couple of ways. Some processors can be put to 'sleep' for a period of time. In this state, processors draw hardly any current. Hence, if the program only requires 10% of the instruction cycles, we can put the processor to sleep for 90% of the time, saving almost 90% of the power. The second option is to reduce the clock-speed of the processor. A reduced clock saves power because almost all power is consumed on clock edges. However, an additional saving can be made when the voltage to the chip is lowered. At a lower clock speed, processors can, in theory, run on a lower voltage. The power consumed is $O(V^2)$ (where V is the voltage), hence an additional reduction in power can be made using this strategy.

In order to show the difference between the power consumed in different processor architectures, Table 1 shows five example processor architectures, each with their power consumption, peak performance, the number of seconds that this system would run on a watch battery (1.5 V $\times$ 18 mah = 0.027 Wh) and a camcorder battery (7.2 V $\times$ 1600 mah = 11.5 Wh), and the number of instructions that the processor can execute in that time. Comparing instructions between processors is doubtful at best – a double precision floating point operation on a Pentium is harder than a 8-bit integer addition on a PIC, and one will need many PIC instructions to perform a 64 bit division. Still, the figures give a clear indication that one can expect between 4 and 160 billion instructions on a watch battery, and between 2 and 70 trillion instructions on a camcorder battery. So it is crucial that the number of instructions is minimised.

In some cases changing the types of instructions used will lead to an overall reduction in the number of instructions executed. Many mobile processors, such as the StrongARM series, do not contain a floating point unit. Any floating point operations are emulated in software. These operations can be replaced by fixed point operations of the desired accuracy at compile-time eliminating the need to execute emulation code, improving power efficiency.

In addition to compile time optimisations that reduce the number of instructions, we can employ run time optimisations that achieve the same goal. The trade-off that one has to make

Device characteristics			Watch battery		Camcorder batt	
Name	Power	'MIPS'	time	Gins	time	Tins
Cray T3D	64 KW	800000	15 ms	12	0.64 sec	5
Pentium IV	50 W	2000	2 s	4	14 min	2
StrongARM	1 W	200	90 s	19	12 hrs	8
PIC	2.4 mW	4	11 h	162	200 days	70
Low freq PIC	60 μW	0.032	450 h	52	22 years	22

Table 1: Number of instructions that can be executed. The figures should be read with caution, as the notion of an instruction is rather ill-defined.

is similar to the trade-off that applies when developing a just-in-time compiler. A just-in-time compiler only makes sense if the time needed for compilation is less than the time saved by faster execution. Similarly, it only makes sense to do a run time program transformation if fewer instructions are spent on the transformation than are saved because of the transformation. The difference between time and instructions is that a just-in-time compiler can be switched on *for free* when the machine is idle, for example because the system is waiting for I/O. However, in the case of pervasive devices, when the processor is idle it is actually saving power, hence, there is no time that we can perform program transformations *for free*.

The simplest metric that we can use to describe power efficiency is the number of instructions executed on a particular processor. This metric has to be used carefully, since not all instructions require the same amount of power.

- On modern high performance processors, parts of the chip are shut down and powered up, depending on what instruction is executed. This powering down is essential as it stops the chip from melting, but it results in a variable power consumption, where a floating point addition may take more power than a bitwise add.

- An instruction, such as add, may use more or less power depending on the operands of the instruction. Adding '0' to '0' may be less work than adding '-1' to '-1'. This difference in power has been shown to be so big that power traces of processors have been successfully used to reverse engineer the data that they operate on [15].

- Between two different processors, instructions are incomparable. Each processor has its own instruction set, and its own computational model. A RISC processor may require more instructions than a CISC, yet use less power.

Having said that, if we run the same program twice on the same processor with and without program transformation, then the number of instructions executed will give us an important insight in the power consumed, especially on simple processors (such as a PIC) that do not have circuitry to switch parts of the chip on and off.

3.2 Communication

Communication is the other major power use in a mobile system. Transmission of data to the network is an intensive energy drain and needs to be minimised in order to provide a reasonable battery life. Reception consumes less energy than transmission in a mobile system, but is still a significant usage. The design of wireless network protocols is dominated by the need to eliminate unnecessary or inefficient communications. For example, in 802.11 devices are multiplexed onto different time slots and only listen for messages during their time-slice. This design eliminates broadcast performance in exchange for adequate client-server performance. Both cannot be accommodated without raising the power consumption significantly.

Given the design choice of how to distribute the activity over various parts of the system, there is a trade-off between performing a computation on a battery-operated processor, or transferring the data to a grid-connected processors, and reading the result back. Communicating the data may cost less power than performing the actual computation.

Communication over an 802.11 network costs in the order of 1W for 1 Mbyte/s (assuming a 5.5Mbs transfer over an 11Mbs link and 200 ma power consumption at 5 V). Radiometrix devices require 100 mW for 16 Kbyte/s. Combined with the data of Table 1 we observe that sending 1 Kbyte of data is equivalent, in terms of power, to executing between 10^6 and 10^7 instructions. Because the relative power consumption of communication and processing is changing all the time, we will initially use the number of bytes transmitted as a measure for the power consumed. When a trade-off between communication and processing is to be made one has to consider how to combine the two metrics into a single one.

In addition to the resource considerations, a number of other aspects in communication must be taken into account when we examine the case studies. The reliability of software for pervasive systems is an important issue to deal with. For example, pervasive systems are expected to have a high degree of communication with other devices, and the detection and handling of errors in communication would become an increasingly important issue. As device-to-device communication increases, so does the need for security. Unintended exposure or perhaps fabrication of information and interruption or interception of communication are security risks that pervasive systems must handle. Verification and correctness analysis of programs or program designs could be a way of ensuring confidentiality, integrity and availability of pervasive systems.

3.3 Memory Footprint

Memory capacities on desktop systems have reached the stage where it is no longer necessary to minimise the memory usage of most applications. Even as the density of storage continues to increase, there is still a trade-off in a portable device that attempts to minimise the memory capacity of the device, to reduce the number and size of its components, and to reduce power consumption. Unlike a desktop system, using a hard-drive to provide virtual memory is not a viable alternative. Hard drives are sensitive to vibration and movement, and large and power hungry compared to solid state memory.

There are several different aspects of an application's memory footprint that can be reduced by program transformations. The amount of memory needed to store the program code can be reduced by eliminating instructions that are unnecessary in the chosen context. Analysing the libraries that are linked into the program code allows only the necessary parts of the API, that are used, to be included. This optimisation reduces the amount of OS / BIOS / ROM support that the application requires. As an extreme example, one could specialise the Linux kernel to exclude modules that are not required by the target applications.

On the other hand, program transformations may increase the amount of memory required. As an example, making a number of specialised versions of a function requires memory to store the specialised versions. This gives us a trade-off between the memory footprint and the number of instructions executed. The metric that we propose to use the number of bytes required for executing the program. The memory requirements may vary over time, so one can choose to use the average number of bytes (which is a meaningful metric for power requirements, if one assumes that banks can be powered on and off at will), or the maximum requirements (which is a meaningful metric for the physical footprint).

3.4 The Design Process.

We are also interested in the design process for pervasive devices and applications. Recent high-level techniques in software development deals with some of the same issues mentioned earlier, like reuse of code and reliability of software, but at a higher design level. One example of a high-level technique could be design patterns, which is a way of describing a solution to a recurring design problem in a systematic and general way. Some research has been done on tools that can automatically generate code from design patterns [25], but this remains an area where specialization tools could make a sizable impact. Since software developers for pervasive systems will face the same issues regarding security, reliability and so on, over and over again, design patterns could become a promising way of dealing with the time-to-market issue.

Aspect weaving is another interesting high-level abstraction that could prove useful to software developers for pervasive systems. In aspect weaving, the design and implementation of functionally separate parts of the program, can be handled individually. Tools supporting automatic aspect weaving and optimization are being investigated [3]. For a pervasive system where, for instance, many different devices communicate using the same communication protocols, it would be beneficial to have a verified component that could easily be shared among many software implementations. Development in a high level language shortens the development time but often requires more resources. Specialization of high-level design methods, which has previously received little attention with respect to their resource consumption, is a necessary area to investigate if those methods should be used in actual pervasive systems software.

An equivalent set of requirements for this process would be:

- Low time to market,
- reusable components, and
- debugging and analysis tools to increase application robustness.

High level tools aid in all these issues. The tools can take the form of a language that can be translated efficiently to mobile devices, or of tools that transform and analyze programs.

The pervasive domain is essentially a consumer-led marketplace. This requires companies to be able to deliver a low time to market for their products in order to remain competitive. In order to deliver a range of designs quickly, it is necessary to allow the developer to work on more generic solutions. This would suggest that a high level language that is amenable to automatic analysis is necessary. The development of a flexible tool-set that allows the user to tune the specialisation process will require clear information from the separate analysis passes. The end product produced by the tools will be sold to non-technical users and so must be robust in operation and not require any tuning of parameters. Although this measure is easy to quantify in a metric (the time that it takes a person to design a solution using a particular set of tools), it is difficult to reliably measure this metric. We do not think that we will evaluate this issue quantitatively, but it is an important rationale for the use of specialisation tools.

3.5 Added functionality

User interfaces for mobile devices are constrained by the requirements for portability and small physical size. These make the use of a conventional mouse or keyboard impractical. In order for an interface to be viable while the user is mobile, it must not dominate their attention. We are experimenting with gesture recognition as an input device and audio playback as an output device. Both the headphones and sensor bands fulfil the criteria of being physically small devices that can be used while mobile.

The processing requirements of interpreting and preparing data for these modes of usage are much greater than a conventional textual interface. In this case specialisation can allow a higher fidelity of experience, for example, more channels of audio and a larger database of gestures. This is hard to quantify in a metric.

3.6 Summary on Metrics

There are a series of quantitative metrics that can be applied to systems to measure the effectiveness of our techniques. Care must taken with these metrics though; they do not allow comparisons between different architectures. A metric can only be used to measure a quantitative change in some aspect of a particular architecture, with respect to the performance on that architecture before our techniques are applied. Another reason that care must be taken with these metrics is that there is no definite way to interpret them - for example we can measure the memory size of a program before and after specialisation, but do we measure the size of the program code, the size of the data used, the size of the executable on disk or even the size in memory as it is executing?

There are also a series of qualitative metrics that can be applied to the design of the system. These will in some cases be a matter of personal taste as qualities such as the expressiveness of the language used cannot be measured directly. Other qualitative metrics will be more obvious, but not comparable. We can always see when features have been added to a system - but we cannot compare the value of different possible additions directly.

4 Expected advances in transformation and analysis tools

The expected performance of the ASAP tools provides additional criteria for the selection of case studies. In this subsection, the limitations of current specialization and analysis tools, and the advances expected from the ASAP tools (WP7), are assessed. This analysis provides requirements on case studies, namely that they should provide opportunities for evaluating the tools' capabilities.

4.1 Language coverage

Program specializers and analysers have been implemented for a variety of languages. Most success has been achieved for declarative languages, presumably due to the lack of a hidden computation state. There are various specializers and partial evaluators that operate on mainstream languages. These can be applied to real code written in C and several other languages. Cmix is a freely available partial evaluator written and maintained by the Cmix group at DIKU [1]. It will handle arbitrary ANSI C programs although the quality of the results obtained will depend on which language features have been used. Tempo is another freely available partial evaluator for C [7].

The target language for ASAP is a Constraint Logic Programming (CLP) language. Thus wide language coverage is not itself a goal of ASAP, but within the CLP context there are several aspects to consider. Below, some of the capabilities and limitations of current tools are listed. Task 7.1 will extend the capabilities of the tools in these respects. The case studies should include code incorporating these more advanced features and extensions in order to evaluate the tools' capabilities.

1. Modular Systems: current tools handle mainly single module programs, whereas analysis of multi-module applications is essential.

2. Meta-programs: current tools handle meta-programs (which are not in principle any different from other programs) but special consideration of efficient representation is needed for some object language constructs, such as constraint stores, environments, substitutions and object language abstract syntax.

3. Advanced control features: the Ciao Prolog environment supports a number of advanced features for controlling program execution, such as explicit concurrency, parallel execution, execution rules alternative to the standard one (e.g., rules other than depth-first search, such as iterative deepening and Andorra-like rules). Analysis and transformation tools should handle such features.

4. Language extensions such as functional, higher-order and object-oriented constructs are supported by Ciao Prolog, and should be handled by the tools.

5. Non-logical extensions (e.g. cut, side-effects). The key issue in treatment of such features is safety. Precise treatment would be a significant challenge. Analysis and transformations should preserve the operational behaviour of these features, but is not expected to perform any deep semantic manipulations.

4.2 Usability

The adoption of tools by developers depends both on their effectiveness and their usability. "Usability" covers several aspects, including user interface, automation, and integration with the development environment.

Currently a hurdle to wide-spread adoption of specialization is that tools are stand-alone, have relatively low-level user-interfaces, while either the target program may require annotation or the user is required to select a suitable control strategy for the specializer. In short, effective use of a specializer may require an expert in specialization.

It is possible that effective use of specialization tools will require applications to be written in a certain style - separation of static (known at compile time) and dynamic (known at run time) arguments is an example. The limitations on which programs can be automatically and efficiently specialised are not syntactic, as most language features can be analysed. The difficulty arises from how those features are used - it is possible to write a program that would confuse a specializer without the use of arrays, pointers and aliasing. The case studies will help to evaluate coding styles and provide appropriate guidance to developers.

Finally, understanding and exploiting of the output of program transformers and analysers is a critical issue in their usability.

4.3 Precision, Termination and Global Control

Advances in the state of the art in program specialization and analysis have been made over the past decade, mainly with respect to precision. In the case of specialization, precision means the ability to exploit the static input more fully, while in analysis, precision means the use of abstractions that describe the concrete behaviour of a program more accurately.

Advances in precision will be studied in ASAP (WP3) and the case studies should provide opportunities for testing and evaluating these advances as well as the trade-offs of precision against complexity.

A related problem is guaranteeing the termination of the specialization process. Closely tied to the problem of guaranteeing termination is the problem of code explosion. An aggressive specialization will attempt to remove as much run-time computation as possible. This can lead to a code-explosion as the residual program contains many copies of procedures with specialized parameters or loops that have been unrolled to the maximum extent. In the worst case this can actually cause a failure to terminate if the specialiser attempts to unroll infinite loops. Programs that manipulate large data-sets present a set of tricky problems. One result can be code-explosion as the data-set is integrated into the residual program. Another problem can be caused in the presence of language features such as arrays and structures is that there is no textual description of the data. This makes it difficult to lift the textual description into the residual program. Solutions to this problem lie in analysis of the termination behaviour of a program as part of a binding-time analysis of the program, and in controlling the numbers of versions of program parts that are generated (polyvariance). Automatically generated control guaranteeing termination would be a major step in the complete automation of specialization.

4.4 Resource-orientation

Most current specializers aim to reduce the run-time of a program. For specialization of pervasive system software, a greater variety of resources needs to be considered. Optimization could be performed with respect to number of instructions processed, memory consumption (internal or external storage), communication, and platform-dependent features (such a specific instructions). These issues and their impact on the tools will be investigated in WP4. Suitable case studies should be selected, such that performance with respect to various resources can be measured.

4.5 Speed and scalability

The efficiency of the tools themselves, while less important for prototype tools, is a major factor in their widespread adoption in the longer term. The complexity of specialization and analysis algorithms will be studied. Case studies should be of sufficient complexity (or be parameterised so that complexity can be varied) in order to study efficiency empirically as well as analytically.

In the case of run-time, on-the-fly code generation using specialization, efficiency is particularly crucial. Techniques such as generating extensions can move part of the complexity to compile-time. A case study involving run-time specialization would provide a vehicle for evaluating the effectiveness such techniques.

4.6 Extra challenging features in case studies

An efficient specialisation requires a good analysis of the original program. This can be complicated by language features that are difficult to analyse such as pointers. Specialisation requires knowledge of variable values at specialisation-time - in the presence of aliasing it can be difficult or even intractable to know which variable is being aliased. Another problem with aliasing is that side-effects (when two parts of a program both independently modify global state) can be disguised or hidden, resulting in an incorrect program transformation so that the resultant program is functionally different from the source program.

The output from a specialiser can be quite inefficient compared to the output of a standard compiler. A specialiser produces very low-level control constructs within the produced code. These are typically chains of goto statements rather than hierarchical control constructs. This output is not particularly readable; this is one of the reasons why the user of a specialiser has to be an expert in the theory of specialisation to understand what has been produced. It also makes it difficult to verify that the specialization process has achieved what the programmer expected. This makes the debugging of specialised programs very difficult.

A standard compiler will produce code that is better structured and optimised. There is also a lack of common algebraic transformations within a specialiser, this means that it cannot take advantage of common optimisations (e.g. the associativity of addition and multiplication). This last issue can be overcome if the specialiser is performing a source-to-source transformation in the same language. The output will then be passed to a standard compiler which will optimise the code.

As we apply specialisation to the domain of pervasive systems we are transforming code that is intended to execute over a very long time-frame. These devices are going to be embedded into the environment and will operate continuously. It may be that they have internal components that are static over short time frames but dynamic over their lifetimes. Current binding time analysers are not capable of recognising this granularity and would limit the aggressiveness of specialisation that can be applied to the system.

4.7 Analysis of typical pervasive software

Standard program analysis techniques may be of limited applicability to the software in pervasive systems. This is due to the assumption that traditional software is algorithmic; it has well defined inputs that are available before execution commences, it executes for a finite length of time and then produces well defined outputs. This assumption does not hold for the software in a pervasive system.

Firstly, pervasive software does not have inputs available prior to commencing execution. As we have detailed in Section 2, a pervasive system does not have a traditional textual interface, so parameters are not passed to the program for execution. Instead, a pervasive system interacts with its environment and input occurs over the lifetime of the program. This will make a binding time analysis of which variables are static and which are dynamic more complex. Depending on the programming model used we can see two methods of accepting input during execution. The program can either read data from an external source during execution (e.g. pipe, channels or streams) or conditional code within the program can be based on the current state of an external interface. For lower level systems the conditional style is more likely and will prove harder to analyse accurately.

Secondly, pervasive software does not have a finite lifespan. It is intended to continue execution indefinitely to provide a service that is embedded within the environment. This means that the software is designed to loop forever rather than to terminate. This suggests that analysis and specialization tools need to be based on a richer (and thus more complex) semantics than usual, allowing states that are not normally observable to be analyzed.

4.8 Summary: Case Study requirements for evaluating advances in tools

Arising from the preceding discussion, a number of desirable aspects of case studies are now listed. These aspects cover features that support the evaluation of the ASAP tools.

1. Language: Ciao Prolog, using multiple modules, covering a wide range of language features and extensions. Note that Ciao Prolog code can be used to emulate other languages, hence case studies involving high-level abstract descriptions (such as process algebras) or low-level abstract machines can be included.

2. Size and complexity: Ideally the size of case study problems should be variable (e.g. a parametrised problem or a study involving a variable number of components), so that complexity and scalability of the tools can be assessed.

3. Resource-orientation: the case studies should make use of a number of different (limited) computation resources such as memory, processors, communication or platform-dependent features (e.g. number of registers).

4. Programming style: algorithms in an interpretive style, handling (streams of) data, possibly perpetual processes. Building of internal data structures which may grow and shrink presents many challenging problems from the point of view of precision, and termination.

   Algorithms which are instances of generic structures or patterns provide opportunities for investigating the application of tools using aspect weaving or design pattern development.

5. Security and safety aspects: case studies with strong requirements on safety and security (possibly incorporating run-time checks) provide evaluation for the use of tools for verifying and analysing run-time behaviour (WP6).

5 Example case studies

In this section we present two examples, and how they fit the requirements outlined earlier. Firstly, the Bristol wearable group has developed an ultrasonic location system [21, 17], in the frame of the Equator project [8]. The most advanced version uses a Kalman filter in estimating its position, where sensor data and update position information are iteratively interpreted by the filter. Secondly, an example of a high-level language called Python, whose main interpreter (i.e.

CPython) has been ported to many platforms, is presented. This example shows, among other things, the early experiences of the Soton team dealing with the porting of Python to pre-Linux handhelds.

5.1 Kalman Filter

Kalman filters are used in many applications that interpret sensor data, and they can be found in devices such as GPS receivers [9]. The Kalman filter is one of the most compute intensive parts of these devices, and its presence is one of the reasons why GPS receivers are battery hungry. The Kalman filter consists of the following operations:

$$\hat{x} = Ax \tag{1}$$
$$\hat{P} = APA^T + Q \tag{2}$$
$$\hat{z} = H\hat{x} \tag{3}$$

$$K = \hat{P}H^T(H\hat{P}H^T + R)^{-1} \tag{4}$$
$$x = \hat{x} + K(z - \hat{z}) \tag{5}$$
$$P = (I - KH)\hat{P} \tag{6}$$

Q, R, A, and H are matrices. The P, Q and A matrix are $N \times N$ (where N is the number of state variables), the R matrix is $M \times M$ (where M is the number of input signals), the others are $N \times M$. There are two issues that require optimisation. One is the sequencing of matrix operations, the other is the sparseness of matrices.

5.1.1 Sequencing of Matrix Operations

When observing Equations 1 to 6, one sees that the Kalman filter regularly requires transposed matrices. Implementing a transpose operation naively requires the creation of a new matrix which is a transposed version of the original. However, it is much cheaper to transpose the matrix while performing the next or the previous operation. So when computing $O = A + B^T$ we can either first calculate $C = B^T$ and then $O = A + C$, or we can in one C function implement a matrix addition with a transposed second argument:

```
void matrix_add_transpose( MAT o, MAT a, MAT bt ) {
  int i,j ;
  for( i=0 ; i<N ; i++ ) {
    for( j=0 ; j<N ; j++ ) {
      o[i][j] = a[i][j] + bt[j][i] ;
    }
  }
}
```

Note that the indices to `bt` have been swapped, and that we add `bt[j][i]` to `a[i][j]`. Indeed, we can implement a library of matrix operations, and create `transpose` versions for each operation, with transposed first, second or output arguments. This is an error prone process and ideally these versions are made on demand by the compiler.[1],

5.1.2 Sparse matrices

The second optimisation concerns the sparseness of some of the matrices. In many cases, the A, Q and R matrices are sparse, some of them may even be diagonal matrices. When working out the equations, multiplying, say, A by Q would require N^3 multiplications. All except for N of those multiplications will end up with a zero-value. If A and Q are diagonal matrices, then a specialised version to multiply A and Q will be:

```
void matrix_mult_AQ( MAT o, MAT a, MAT q ) {
  int i ;
  for( i=0 ; i<N ; i++ ) {
    o[i][i] = a[i][i] * q[i][i] ;
  }
}
```

It can be simplified further – since A and Q only have values on the diagonal, there is no reason to store them in a matrix, they could be stored in a vector of length N instead. Only the diagonal values of o are assigned to in the function `matrix_mult_AQ`. if o is going to be used as a matrix, then the non diagonal elements should be set to 0, but a transformation tool could propagate knowledge about the zero values of o, and simply leave those values uninitialised. In that case o could be stored as a vector.

Again, the specialised versions should be made automatically.

[1] Since suggesting the case study, a bug has been uncovered in a hand crafted specialised function for matrix subtraction that calculates $(I - x)$ (the identity matrix minus some other matrix). This problem only came to the surface on a code-audit, for the Kalman filter up to a degree manages to correct this error. After correction, the quality of the filtered data improved dramatically. This highlights the need for automated optimisations in favour of hand-crafted specialisations.

Note that this optimisation is not restricted to Kalman filters, but will work on any algorithm that relies on matrix operations.

Sparse matrix representation has been the subject of many papers (for example [23, 22]) in the starting days of computer science, when programmers needed to store 1000 by 1000 matrices in 32 Kbyte of memory. In the case of the Kalman filter, the problem is slightly more down to earth. The matrices are smaller (10 by 10), and the gain is in reducing processing power rather than reducing memory space.

5.1.3 Criteria

We have Kalman filters available in a variety of sizes, from just a two state variables up to 8 state variables, and up to 4 data inputs. The core of the Kalman filter is easy to implement in any language (we have a slightly optimised C implementation available) — it just requires a matrix library. It can execute on any machine.

A mathematical definition of the Kalman filer is also available, and is well understood.

Initial measurements suggest that 40% of the multiplications in the matrix library are used to multiply by zero, 10% multiply by one, and 70% of the additions add zero. This indicates that we should be able to reduce the number of operations performed by 50%. In addition, constants can be propagated as part of the specialisation, reducing overhead in calculating matrix indices.

Of particular interest is that the core of the Kalman filter is normally implemented as a library module, which is then used with appropriate definitions of H, A, Q and R. So once an automated program transformation system can deal with the core of the Kalman filter, it can be applied to any program using the filter. Depending on the sparseness of the matrices in the particular domain, the program transformation will be able to save more or less instructions.

5.2 Python Ports

On modern Linux-based handhelds, ports of CPython are reasonably complete and easy to maintain. The available interpreters are thus close — in terms of available features — to the “official” one running on workstations.

Among the few tradeoffs one could note that the handheld-based interpreters are typically compiled with an option that removes the documentation strings that are normally available in an interactive Python environment, in order to save memory; but the available Linux environments are pretty complete and don’t require much more tuning of the CPython source.

Most of the large range of Python applications available on workstations should readily run on handhelds. Among the most interesting ones are the network-oriented ones. There are for example several Wiki implementations in Python, which may provide a convenient way to share and edit information in an informal way among a group of persons. Multiplayer games are another immediate example.

While scripting languages are meant to easily write “glue” code between applications, Python also offers high-level programming. For example, a custom web servers providing dynamic information is a matter of two pages of code; this makes the handheld publicly consultable using any web browser, e.g. to export the public part of the owner’s agenda, or some real-time data gathered by a data acquisition device. More information can be found at [19, 11, 10, 20].

Requirements:

1. Memory limitation: total memory limits the size of the applications that can be considered. However with a common 32MB of disk+RAM space and applications in compact pseudo-code (as opposed to native machine binaries) this limit is high. Using the same interpreter for several applications lowers the weight of the interpreter.

2. CPU speed: hundreds of MHz are plenty enough even given the high overhead of interpreters for large classes of applications.

3. Battery savings: compact pseudo-code may translate into less actual memory accesses, at least after the interpreter has started up. This point probably depends quite a lot on the application.

5.2.1 A networked game on the iPAQ

Porting a networked game (`http://bub-n-bros.sourceforce.org`) to a wireless-enabled iPAQ was surprisingly straightforward and done within a day. All the actual problems were not related to programming restrictions; the speed and available memory were quite sufficient. The troubles where related to the different user interface, which lies outside the scope of this project, but which we summarize here because they are noteworthy and interesting:

1. The screen is high-color (16 bits per pixel) but its resolution of 320x240 needs to be kept in mind, specially for user-interface-intensive applications like games. In the present case just pre-down-sampling all graphics by a factor of 2 proved to be an appropriate solution.

2. The “software keyboard” is not practical for games. Moreover, the few physical keys suffer from a controller that is not able to detect if multiple keys are pressed at the same time. Given these restrictions, a custom input method has been designed: the gameplay was

determined by mouse movements alone, which the user controls by moving the pointing pen around the screen.

3. The machine hardware can synthesize 16-bits-sampled sounds. Some conversion was required from the existing game, which outputted 8-bit sounds (which on the present version of Linux was synthesized anyway and resulted in raw noise).

5.2.2 Other ports

Python has been ported to pre-Linux handhelds as well, which typically offer much less processing power and memory, and an environment quite different from the Unix standards.

The most advanced port to date is *Pippy* for the PalmOS. It typically runs on devices like today's low-end Palm i705 (8MB RAM + 4MB ROM). It has even been tested with some success on the Palm III, a 2MB RAM device. The Python Virtual Machine and libraries can occupy as little as 191 KB of storage memory and will execute in less than 64 KB of heap space.

The Palm i705 offers a 160x160 monochrome interface with no standard command-line environment. Designing such a command-line interface is one of the few things the porting team had to do from scratch.

The port is based on an older stable compact version of the interpreter. As always, every new version of the workstation-based interpreter is a bit larger, so is a bit more difficult to squeeze down on a memory-constrained device.

A number of language features were removed:

- Floating-point and complex numbers;
- Filesystem-based file I/O (the Palm has a different custom way of handling "files");
- Interactive documentation strings.
- The parser and the compiler are optional; for non-interactive sessions we can download precompiled bytecode files directly to the Palm.
- Most of the Python standard library cannot be included by default for size reasons; needed modules can be selectively added.

As most of the Posix-oriented interfaces have to be removed, some custom Palm-specific modules have been added. This lowers the compatibility with the existing Python code and forces programmers to learn a new, Palm-specific interface.

The port is also less useful than a Linux-based one because the interpreter cannot run in the background to provide services. Its main interest seems to lie essentially in that it offers the possibility to write small personal scripts for day-to-day tasks.

Among the difficult problems encountered during the port is the fact that the Palm processor stack is fixed and small (sometimes as low as 8Kb). In the normal Python interpreter written in C, recursive Python calls also mean recursive C calls, so that recursive programs can easily overflow the stack. This is more a problem in high-level languages than, say, in C, because a number of high-level languages tend to favour recursion (and not just tail-recursion). A possible workaround is to write the interpreter in a non-recursive fashion, as Stackless aims to do for workstations.

Memory management in general was a main concern of Pippy. It seems that it is a much more important concern on the Palm than on workstations, and not just for the fact that it is a limited resource. For example, moving read-only data out of the dynamic heap into special storage areas of the Palm was a net win.

Most importantly, the interpreter is *malloc()*-bound; i.e. most of the operations require allocating and reallocating small (reference-counted) objects on the heap. Replacing the Palm's default *malloc()* implementation with a sophisticated custom implementation (Doug Lea's) proved to be a huge win, even though the extra code makes the interpreter larger. It seems that in general never trusting any *malloc()* implementation whatsoever to perform reasonably well for a large number of small objects is the path to go. (Note that even CPython introduced a custom memory allocator in version 2.1.)

The previous paragraph does not directly apply to garbage-collected systems, although one can guess that finely tuning a garbage collector is equally more difficult on a handheld than on a workstation. Further information can be found in [6, 29, 4, 5, 26, 16].

6 Summary of Requirements for case studies

There are many practical requirements that we need to consider in addition to the measurable objectives listed in the previous section. For each candidate case study we must answer the set of questions outlined below.

6.1 Implementation

In order for the case study to be of practical use we need something that is either implemented and tested in the field, or something for which a design is available. If the case study has been implemented we need to know:

- Which language it is written in,
- whether tools exist that can specialise this language, and
- whether we need access to any necessary hardware.
- Is the implementation written in an interpretative style? Does the implementation consist of never ending processes?

If the case study has not been implemented, but a complete design is available, then we need to know whether we can model the case study in a language for which specialisation and analysis tools are available in ASAP, for example CIAO/Prolog.

6.2 Specification

Once program transformations have been applied to a case study, we need to ensure that the case study fulfils its specification. In addition, if the specification is executable, it can be used as a starting point for program transformations. So, if a high level specification of the problem is available, then we need to know:

- Whether it is an executable specification,
- which language it is written in, and
- whether tools exist that can specialise this language.

6.3 Costs / Benefits

For each case study we will have to perform a cost / benefit analysis. In particular, we need to find out whether it is worth investing time into a particular case study, in terms of what we expect to learn from it or prove with it. The value of the case study can be determined by answering the following questions:

- Is the problem size variable so that scalability can be assessed?
- What are the success criteria, in terms of the metrics presented in Section 3, and does it employ more than just one resource?
- What benefits do we expect from program analysis and specialisation tools, in terms of the success criteria?

- What benefits do we expect in terms of improving program analysis and specialisation?
- How much effort is it to understand, run, model and analyse the case study?
- What information sources are available for the case study, for example in the form of Web-pages, research papers?

Based on those requirements, we can select a useful set of case studies.

References

[1] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, DIKU, 1994.

[2] Stavros Antifakos, Florian Michahelles, and Bernt Schiele. Proactive Instructions for Furniture Assembly. In *Proceedings of the The Fourth International Conference on Ubiquitous Computing (UbiComp 2002)*, Göteborg, Sweden, September 2002.

[3] Jason Baker and Wilson C. Hsieh. Runtime aspect weaving through metaprogramming. In *First International Conference on Aspect-Oriented Software Design*, pages 86–95. ACM, 2002.

[4] Jeff Bauer. Python CE (for Windows CE). In *Proceedings of the 7th International Python Conference*. http://www.foretec.com/python/workshops/1998-11/demosession/bauer.html, November 2998.

[5] Duncan Booth. Python for Epoc Systems. http://dales.rmplc.co.uk/Duncan/PyPsion.htm.

[6] Jeffery D. Collins. Pippy. http://pippy.sourceforge.net/.

[7] C. Consel, L. Hornof, R. Marlet, G. Muller, S. Thibault, and E.-N. Volanschi. Tempo: specializing systems applications and beyond. *ACM Computing Surveys*, 30(3es), 1998.

[8] Equator. The EQUATOR Interdisciplinary Research Collaboration. http://www.equator.ac.uk/.

[9] Mohinder S. Grewal, Lawrence R. Weill, and Angus P. Andrews. *Global Positioning Systems, Inertial Navigation, and Integration*. Wiley, Inc., 2001.

[10] James Henstridge. PyGTK, bindings for the GTK graphical user interface components. http://www.daa.com.au/∼james/software/pygtk/.

[11] Jürgen Hermann. MoinMoin, a Wiki in Python. http://moin.sourceforge.net/.

[12] B. Horrigan. The home of tomorrow: 1927-45. Imagining Tomorrow: History Technology and the American Future, Cambridge, Mass: MIT Press, 1987.

[13] Intel. *Intel StrongARM SA-1100 Microprocessor Developer's Manual*. Intel, Apr 1999.

[14] C. D. Kidd, R. Orr, G. D. Abowd, C. G. Atkeson, I. A. Essa, B. MacIntyre, E. Mynatt, T. E. Starner, and W. Newstetter. The aware home: A living laboratory for ubiquitous computing research. Second International Workshop on Cooperative Buildings - CoBuild'99, 1999.

[15] P. Kocher, J. Jaffe, and B Jun. Differential power analysis. In *Advances in Cryptology, CRYPTO '99*, pages 388–397, 1999.

[16] Doug Lea. Doug Lea's memory allocator. http://g.oswego.edu/dl/html/malloc.html.

[17] Mike McCarthy and Henk Muller. No pingers: Ultrasonic indoor location sensing without rf synchonisation. Technical Report 003-004, University of Bristol, Department of Computer Science, May 2003.

[18] Microchip. *PIC16F84A Data Sheet: 18-pin Enhanced FLASH/EEPROM 8-bit Microcontroller; Document DS35007B*. Microship Technology Inc, 2001.

[19] Edward Mueller. Python on the iPAQ. http://www.handhelds.org/minihowto/python-ipaq.html.

[20] Python. The standard library's web protocols. http://www.python.org/doc/current/lib/internet.html.

[21] Cliff Randell and Henk Muller. Low cost indoor positioning system. In Gregory D. Abowd, editor, *Ubicomp 2001: Ubiquitous Computing*, pages 42–48. Springer-Verlag, September 2001.

[22] D.J. Rose and R.A. Willoughby, editors. *Sparse Matrices and Their Applications*. Plenum Press, New York, 1969.

[23] A Ruhe. Sor-methods for the eigenvalue problem of large sparse matrices. *Math.Comp*, pages 695–710, 1974.

[24] Federico Sartor. Dow Jones Tuesday (April 9), 2003.

[25] Ulrik Schultz, Julia L. Lawall, and Charles Consel. Automatic program specialization for Java. *ACM Transactions on Programming Languages and Systems*, 25(4):452–499, 2003.

[26] Christian Tismer. Stackless.
http://www.stackless.com/.

[27] Andrew Turley. Embedded processors by the numbers. *Embedded Systems Programming*, 12(5), May 1999.

[28] Roy Want, Andy Hopper, Veronica Falcao, and Jonathon Gibbons. The active badge location system. *ACM Transactions on Information Systems*, 10(1):91–102, January 1992.

[29] Brian Ward. A Practical Scripting Environment for Mobile Devices. In *Usenix Annual Technical Conference*, pages 43–54.
http://www.usenix.org/events/usenix01/freenix01/full_papers/ward/ward_html/,
2001.