Run-Time Dynamic Linking for Reprogramming Wireless Sensor Networks

Adam Dunkels, Niclas Finne, Joakim Eriksson, Thiemo Voigt
Swedish Institute of Computer Science, Box 1263, SE-16429 Kista, Sweden
adam@sics.se, nfi@sics.se, joakime@sics.se, thiemo@sics.se

Abstract

From experience with wireless sensor networks it has become apparent that dynamic reprogramming of the sensor nodes is a useful feature. The resource constraints in terms of energy, memory, and processing power make sensor network reprogramming a challenging task. Many different mechanisms for reprogramming sensor nodes have been developed ranging from full image replacement to virtual machines.

We have implemented an in-situ run-time dynamic linker and loader that use the standard ELF object file format. We show that run-time dynamic linking is an effective method for reprogramming even resource constrained wireless sensor nodes. To evaluate our dynamic linking mechanism we have implemented an application-specific virtual machine and a Java virtual machine and compare the energy cost of the different linking and execution models. We measure the energy consumption and execution time overhead on real hardware to quantify the energy costs for dynamic linking.

Our results suggest that while in general the overhead of a virtual machine is high, a combination of native code and virtual machine code provide good energy efficiency. Dynamic run-time linking can be used to update the native code, even in heterogeneous networks.

Categories and Subject Descriptors

C.2.4 [**Computer Communication Networks**]: Distributed Systems—*Network Operating Systems*

General Terms

Design, Experimentation, Measurement, Performance

Keywords

Wireless sensor networks, Embedded systems, Operating systems, Dynamic linking, Virtual machines

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SenSys'06, November 1–3, 2006, Boulder, Colorado, USA.
Copyright 2006 ACM 1-59593-343-3/06/0011 ...$5.00

1 Introduction

Wireless sensor networks consist of a collection of programmable radio-equipped embedded systems. The behavior of a wireless sensor network is encoded in software running on the wireless sensor network nodes. The software in deployed wireless sensor network systems often needs to be changed, both to update the system with new functionality and to correct software bugs. For this reason dynamically reprogramming of wireless sensor network is an important feature. Furthermore, when developing software for wireless sensor networks, being able to update the software of a running sensor network greatly helps to shorten the development time.

The limitations of communication bandwidth, the limited energy of the sensor nodes, the limited sensor node memory which typically is on the order of a few thousand bytes large, the absence of memory mapping hardware, and the limited processing power make reprogramming of sensor network nodes challenging.

Many different methods for reprogramming sensor nodes have been developed, including full system image replacement [14, 16], approaches based on binary differences [15, 17, 31], virtual machines [18, 19, 20], and loadable native code modules in the first versions of Contiki [5] and SOS [12]. These methods are either inefficient in terms of energy or require non-standard data formats and tools.

The primary contribution of this paper is that we investigate the use of standard mechanisms and file formats for reprogramming sensor network nodes. We show that in-situ dynamic run-time linking and loading of native code using the ELF file format, which is a standard feature on many operating systems for PC computers and workstations, is feasible even for resource-constrained sensor nodes. Our secondary contribution is that we measure and quantify the energy costs of dynamic linking and execution of native code and compare it to the energy cost of transmission and execution of code for two virtual machines: an application-specific virtual machine and the Java virtual machine.

We have implemented a dynamic linker in the Contiki operating system that can link, relocate, and load standard ELF object code files. Our mechanism is independent of the particular microprocessor architecture on the sensor nodes and we have ported the linker to two different sensor node platforms with only minor modifications to the architecture dependent module of the code.

To evaluate the energy costs of the dynamic linker we implement an application specific virtual machine for Contiki together with a compiler for a subset of Java. We also adapt the Java virtual machine from the lejOS system [8] to run under Contiki. We measure the energy cost of reprogramming and executing a set of program using dynamic linking of native code and the two virtual machines. Using the measurements and a simple energy consumption model we calculate break-even points for the energy consumption of the different mechanisms. Our results suggest that while the execution time overhead of a virtual machine is high, a combination of native code and virtual machine code may give good energy efficiency.

The remainder of this paper is structured as follows. In Section 2 we discuss different scenarios in which reprogramming is useful. Section 3 presents a set of mechanisms for executing code inside a sensor node and in Section 4 we discuss loadable modules and the process of linking, relocating, and loading native code. Section 5 describes our implementation of dynamic linking and our virtual machines. Our experiments and the results are presented in Section 6 and discuss the results in Section 7. Related work is reviewed in Section 8. Finally, we conclude the paper in Section 9.

2 Scenarios for Software Updates

Software updates for sensor networks are necessary for a variety of reasons ranging from implementation and testing of new features of an existing program to complete reprogramming of sensor nodes when installing new applications. In this section we review a set of typical reprogramming scenarios and compare their qualitative properties.

2.1 Software Development

Software development is an iterative process where code is written, installed, tested, and debugged in a cyclic fashion. Being able to dynamically reprogram parts of the sensor network system helps shorten the time of the development cycle. During the development cycle developers typically change only one part of the system, possibly only a single algorithm or a function. A sensor network used for software development may therefore see large amounts of small changes to its code.

2.2 Sensor Network Testbeds

Sensor network testbeds are an important tool for development and experimentation with sensor network applications. New applications can be tested in a realistic setting and important measurements can be obtained [36]. When a new application is to be tested in a testbed the application typically is installed in the entire network. The application is then run for a specified time, while measurements are collected both from the sensors on the sensor nodes, and from network traffic.

For testbeds that are powered from a continuous energy source, the energy consumption of software updates is only of secondary importance. Instead, qualitative properties such as ease of use and flexibility of the software update mechanism are more important. Since the time required to make an update is important, the throughput of a network-wide software update is of importance. As the size of the transmitted binaries impact the throughput, the binary size still can be used as an evaluation metric for systems where throughput is more important than energy consumption.

Scenario	Update frequency	Update fraction	Update level	Program longevity
Development	Often	Small	All	Short
Testbeds	Seldom	Large	All	Long
Bug fixes	Seldom	Small	All	Long
Reconfig.	Seldom	Small	App	Long
Dynamic Application	Often	Small	App	Long

Table 1. Qualitative comparison between different reprogramming scenarios.

2.3 Correction of Software Bugs

The need for correcting software bugs in sensor networks was early identified [7]. Even after careful testing, new bugs can occur in deployed sensor networks caused by, for example, an unexpected combination of inputs or variable link connectivity that stimulate untested control paths in the communication software [30].

Software bugs can occur at any level of the system. To correct bugs it must therefore be possible to reprogram all parts of the system.

2.4 Application Reconfiguration

In an already installed sensor network, the application may need to be reconfigured. This includes change of parameters, or small changes in the application such as changing from absolute temperature readings to notification when thresholds are exceeded [26]. Even though reconfiguration not necessarily include software updates [25], application reconfiguration can be done by reprogramming the application software. Hence software updates can be used in an application reconfiguration scenario.

2.5 Dynamic Applications

There are many situations where it is useful to replace the application software of an already deployed sensor network. One example is the forest fire detection scenario presented by Fok et al. [9] where a sensor network is used to detect a fire. When the fire detection application has detected a fire, the fire fighters might want to run a search and rescue application as well as a fire tracking application. While it may possible to host these particular applications on each node despite the limited memory of the sensor nodes, this approach is not scalable [9]. In this scenario, replacing the application on the sensor nodes leads to a more scalable system.

2.6 Summary

Table 1 compares the different scenarios and their properties. *Update fraction* refers to what amount of the system that needs to be updated for every update, *update level* to at what levels of the system updates are likely to occur, and *program longevity* to how long an installed program will be expected to reside on the sensor node.

3 Code Execution Models and Reprogramming

Many different execution models and environments have been developed or adapted to run on wireless sensor nodes.

Some with the notion of facilitating programming [1], others motivated by the potential of saving energy costs for reprogramming enabled by the compact code representation of virtual machines [19]. The choice of the execution model directly impacts the data format and size of the data that needs to be transported to a node. In this section we discuss three different mechanisms for executing program code inside each sensor node: script languages, virtual machines, and native code.

3.1 Script Languages

There are many examples of script languages for embedded systems, including BASIC variants, Python interpreters [22], and TCL machines [1]. However, most script interpreters target platforms with much more resources than our target platforms and we have therefore not included them in our comparison.

3.2 Virtual Machines

Virtual machines are a common approach to reduce the cost of transmitting program code in situations where the cost of distributing a program is high. Typically, program code for a virtual machine can be made more compact than the program code for the physical machine. For this reason virtual machines are often used for programming sensor networks [18, 19, 20, 23].

While many virtual machines such as the Java virtual machine are generic enough to perform well for a variety of different types of programs, most virtual machines for sensor networks are designed to be highly configurable in order to allow the virtual machine to be tailored for specific applications. In effect, this means that parts of the application code is implemented as virtual machine code running on the virtual machine, and other parts of the application code is implemented in native code that can be used from the programs running on the virtual machine.

3.3 Native Code

The most straightforward way to execute code on sensor nodes is by running native code that is executed directly by the microcontroller of the sensor node. Installing new native code on a sensor node is more complex than installing code for a virtual machine because the native code uses physical addresses which typically need to be updated before the program can be executed. In this section we discuss two widely used mechanisms for reprogramming sensor nodes that execute native code: full image replacement and approaches based on binary differences.

3.3.1 Full Image Replacement

The most common way to update software in embedded systems and sensor networks is to compile a complete new binary image of the software together with the operating system and overwrite the existing system image of the sensor node. This is the default method used by the XNP and Deluge network reprogramming software in TinyOS [13].

The full image replacement does not require any additional processing of the loaded system image before it is loaded into the system, since the loaded image resides at the same, known, physical memory address as the previous system image. For some systems, such as the Scatterweb system code [33], the system contains both an operating system image and a small set of functions that provide functionality for loading new operating system images. A new operating system image can overwrite the existing image without overwriting the loading functions. The addresses of the loading functions are hard-coded in the operating system image.

3.3.2 Diff-based Approaches

Often a small update in the code of the system, such as a bugfix, will cause only minor differences between in the new and old system image. Instead of distributing a new full system image the binary differences, deltas, between the modified and original binary can be distributed. This reduces the amount of data that needs to be transferred. Several types of diff-based approaches have been developed [15, 17, 31] and it has been shown that the size of the deltas produced by the diff-based approaches is very small compared to the full binary image.

4 Loadable Modules

A less common alternative to full image replacement and diff-based approaches is to use loadable modules to perform reprogramming. With loadable modules, only parts of the system need to be modified when a single program is changed. Typically, loadable modules require support from the operating system. Contiki and SOS are examples of systems that support loadable modules and TinyOS is an example of an operating system without loadable module support.

A loadable module contains the native machine code of the program that is to be loaded into the system. The machine code in the module usually contains references to functions or variables in the system. These references must be resolved to the physical address of the functions or variables before the machine code can be executed. The process of resolving those references is called linking. Linking can be done either when the module is compiled or when the module is loaded. We call the former approach pre-linking and the latter dynamic linking. A pre-linked module contains the absolute physical addresses of the referenced functions or variables whereas a dynamically linked module contains the symbolic names of all system core functions or variables that are referenced in the module. This information increases the size of the dynamically linked module compared to the pre-linked module. The difference is shown in Figure 1. Dynamic linking has not previously been considered for wireless sensor networks because of the perceived run-time overhead, both in terms of execution time, energy consumption, and memory requirements.

The machine code in the module usually contains references not only to functions or variables in the system, but also to functions or variables within the module itself. The physical address of those functions will change depending on the memory address at which the module is loaded in the system. The addresses of the references must therefore be updated to the physical address that the function or variable will have when the module is loaded. The process of updating these references is known as relocation. Like linking, relocation can be done either at compile-time or at run-time.

When a module has been linked and relocated the program loader loads the module into the system by copying the

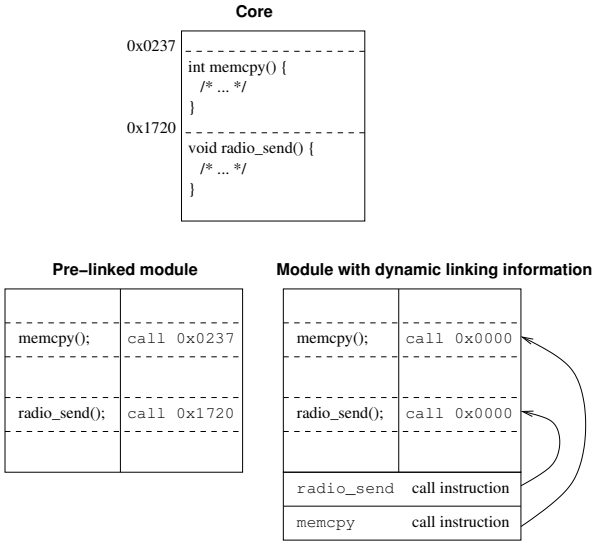

Figure 1. The difference between a pre-linked module and a module with dynamic linking information: the pre-linked module contains physical addresses whereas the dynamically linked module contains symbolic names.

linked and relocated native code into a place in memory from where the program can be executed.

4.1 Pre-linked Modules

The machine code of a pre-linked module contains absolute addresses of all functions and variables in the system code that are referenced by the module. Linking of the module is done at compile time and only relocation is performed at run-time. To link a pre-linked module, information about the physical addresses of all functions and variables in the system into which the module is to be loaded must be available at compile time.

There are two benefits of pre-linked modules over dynamically linked modules. First, pre-linked modules are smaller than dynamically linked modules which results in less information to be transmitted. Second, the process of loading a pre-linked module into the system is less complex than the process of linking a dynamically linked module. However, the fact that all physical addresses of the system core are hard-coded in the pre-linked module is a severe drawback as a pre-linked module can only be loaded into a system with the exact same physical addresses as the system that was to generate the list of addresses that was used for linking the module.

In the original Contiki system [5] we used pre-linked binary modules for dynamic loading. When compiling the Contiki system core, the compiler generated a map file containing the mapping between all globally visible functions and variables in the system core and their addresses. This list of addresses was used to pre-link Contiki modules.

We quickly noticed that while pre-linked binary modules worked well for small projects with a homogeneous set of sensor nodes, the system quickly became unmanageable when the number of sensor nodes grew. Even a small change to the system core of one of the sensor nodes would make it impossible to load binary a module into the system bedcase the addresses of variables and functions in the core were different from when the program was linked. We used version numbers to guard against this situation. Version numbers did help against system crashes, but did not solve the general problem: new modules could not be loaded into the system.

4.2 Dynamic Linking

With dynamic linking, the object files do not only contain code and data, but also names of functions are variables of the system core that are referenced by the module. The code in the object file cannot be executed before the physical addresses of the referenced variables and functions have been filled in. This process is done at run time by a dynamic linker.

In the Contiki dynamic linker we use two file formats for the dynamically linked modules, ELF and Compact ELF.

4.2.1 ELF - Executable and Linkable Format

One of the most common object code format for dynamic linking is the Executable and Linkable Format (ELF) [3]. It is a standard format for object files and executables that is used for most modern Unix-like systems. An ELF object file include both program code and data and additional information such as a symbol table, the names of all external unresolved symbols, and relocation tables. The relocation tables are used to locate the program code and data at other places in memory than for which the object code originally was assembled. Additionally, ELF files can hold debugging information such as the line numbers corresponding to specific machine code instructions, and file names of the source files used when producing the ELF object.

ELF is also the default object file format produced by the GCC utilities and for this reason there are a number of standard software utilities for manipulating ELF files available. Examples include debuggers, linkers, converters, and programs for calculating program code and data memory sizes. These utilities exist for a wide variety of platforms, including MS Windows, Linux, Solaris, and FreeBSD. This is a clear advantage over other solutions such as FlexCup [27], which require specialized utilities and tools.

Our dynamic linker in Contiki understands the ELF format and is able to perform dynamic linking, relocation, and loading of ELF object code files. The debugging features of the ELF format are not used.

4.2.2 CELF - Compact ELF

One problem with the ELF format is the overhead in terms of bytes to be transmitted across the network, compared to pre-linked modules. There are a number of reasons for the extra overhead. First, ELF, as any dynamically relocatable file format, includes the symbolic names of all referenced functions or variables that need to be linked at run-time. Second, and more important, the ELF format is designed to work on 32-bit and 64-bit architectures. This causes all ELF data structures to be defined with 32-bit data types. For 8-bit or 16-bit targets the high 16 bits of these fields are unused.

To quantify the overhead of the ELF format we devise an alternative to the ELF object code format that we call CELF - Compact ELF. A CELF file contains the same information as an ELF file, but represented with 8 and 16-bit datatypes.

CELF files typically are half the size of the corresponding ELF file. The Contiki dynamic loader is able to load CELF files and a utility program is used to convert ELF files to CELF files.

It is possible to further compress CELF files using lossless data compression. However, we leave the investigation of the energy-efficiency of this approach to future work.

The drawback of the CELF format is that it requires a special compressor utility is for creating the CELF files. This makes the CELF format less attractive for use in many real-world situations.

4.3 Position Independent Code

To avoid performing the relocation step when loading a module, it is in some cases possible to compile the module into position independent code. Position independent code is a type of machine code which does not contain any absolute addresses to itself, but only relative references. This is the approach taken by the SOS system.

To generate position independent code compiler support is needed. Furthermore, not all CPU architectures support position independent code and even when supported, programs compiled to position independent code typically are subject to size restrictions. For example, the AVR microcontroller supports position independent code but restricts the size of programs to 4 kilobytes. For the MSP430 no compiler is known to fully support position independent code.

5 Implementation

We have implemented run-time dynamic linking of ELF and CELF files in the Contiki operating system [5]. To evaluate dynamic linking we have implemented an application specific virtual machine for Contiki together with a compiler for a subset of Java, and have ported a Java virtual machine to Contiki.

5.1 The Contiki Operating System

The Contiki operating system was the first operating system for memory-constrained sensor nodes to support dynamic run-time loading of native code modules. Contiki is built around an event-driven kernel and has very low memory requirements. Contiki applications run as extremely lightweight protothreads [6] that provide blocking operations on top of the event-driven kernel at a very small memory cost. Contiki is designed to be highly portable and has been ported to over ten different platforms with different CPU architectures and using different C compilers.

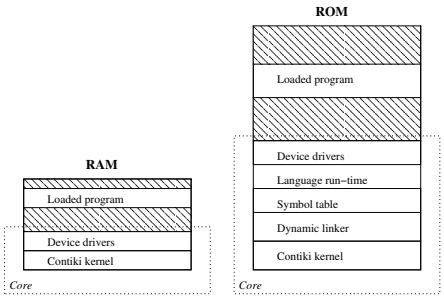

Figure 2. Partitioning in Contiki: the core and loadable programs in RAM and ROM.

A Contiki system is divided into two parts: the core and the loadable programs as shown in Figure 2. The core consists of the Contiki kernel, device drivers, a set of standard applications, parts of the C language library, and a symbol table. Loadable programs are loaded on top of the core and do not modify the core.

The core has no information about the loadable programs, except for information that the loadable programs explicitly register with the core. Loadable programs, on the other hand, have full knowledge of the core and may freely call functions and access variables that reside in the core. Loadable programs can call each other by going through the kernel. The kernel dispatches calls from one loaded program to another by looking up the target program in an in-kernel list of active processes. This one-way dependency makes it possible to load and unload programs at run-time without needing to patch the core and without the need for a reboot when a module has been loaded or unloaded.

While it is possible to replace the core at run-time by running a special loadable program that overwrites the current core and reboots the system, experience has shown that this feature is not often used in practice.

5.2 The Symbol Table

The Contiki core contains a table of the symbolic names of all externally visible variable and function names in the Contiki core and their corresponding addresses. The table includes not only the Contiki system, but also the C language run-time library. The symbol table is used by the dynamic linker when linking loaded programs.

The symbol table is created when the Contiki core binary image is compiled. Since the core must contain a correct symbol table, and a correct symbol table cannot be created before the core exists, a three-step process is required to compile a core with a correct symbol table. First, an intermediary core image with an empty symbol table is compiled. From the intermediary core image an intermediary symbol table is created. The intermediary symbol table contains the correct symbols of the final core image, but the addresses of the symbols are incorrect. Second, a second intermediary core image that includes the intermediary symbol table is created. This core image now contains a symbol table of the same size as the one in the final core image so the addresses of all symbols in the core are now as they will be in the final core image. The final symbol table is then created from the second intermediary core image. This symbol table contains both the correct symbols and their correct addresses. Third, the final core image with the correct symbol table is compiled.

The process of creating a core image is automated through a simple `make` script. The symbol table is created using a combination of standard ELF tools.

For a typical Contiki system the symbol table contains around 300 entries which amounts to approximately 4 kilobytes of data stored in flash ROM.

5.3 The Dynamic Linker

We implemented a dynamic linker for Contiki that is designed to link, relocate, and load either standard ELF files [3] and CELF, Compact ELF, files. The dynamic linker reads

ELF/CELF files through the Contiki virtual filesystem interface, CFS, which makes the dynamic linker unaware of the physical location of the ELF/CELF file. Thus the linker can operate on files stored either in RAM, on-chip flash ROM, external EEPROM, or external ROM without modification. Since all file access to the ELF/CELF file is made through the CFS, the dynamic linker does not need to concern itself with low-level filesystem details such as wear-leveling or fragmentation [4] as this is better handled by the CFS.

The dynamic linker performs four steps to link, relocate and load an ELF/CELF file. The dynamic linker first parses the ELF/CELF file and extracts relevant information about where in the ELF/CELF file the code, data, symbol table, and relocation entries are stored. Second, memory for the code and data is allocated from flash ROM and RAM, respectively. Third, the code and data segments are linked and relocated to their respective memory locations, and fourth, the code is written to flash ROM and the data to RAM.

Currently, memory allocation for the loaded program is done using a simple block allocation scheme. More sophisticated allocation schemes will be investigated in the future.

5.3.1 Linking and Relocating

The relocation information in an ELF/CELF file consists of a list of relocation entries. Each relocation entry corresponds to an instruction or address in the code or data in the module that needs to be updated with a new address. A relocation entry contains a pointer to a symbol, such as a variable name or a function name, a pointer to a place in the code or data contained in the ELF/CELF file that needs to be updated with the address of the symbol, and a relocation type which specifies how the data or code should be updated. The relocation types are different depending on the CPU architecture. For the MSP430 there is only one single relocation type, whereas the AVR has 19 different relocation types.

The dynamic linker processes a relocation entry at a time. For each relocation entry, its symbol is looked up in the symbol table in the core. If the symbol is found in the core's symbol table, the address of the symbol is used to patch the code or data to which the relocation entry points. The code or data is patched in different ways depending on the relocation type and on the CPU architecture.

If the symbol in the relocation entry was not found in the symbol table of the core, the symbol table of the ELF/CELF file itself is searched. If the symbol is found, the address that the symbol will have when the program has been loaded is calculated, and the code or data is patched in the same way as if the symbol was found in the core symbol table.

Relocation entries may also be relative to the data, BSS, or code segment in the ELF/CELF file. In that case no symbol is associated with the relocation entry. For such entries the dynamic linker calculates the address that the segment will have when the program has been loaded, and uses that address to patch the code or data.

5.3.2 Loading

When the linking and relocating is completed, the text and data have been relocated to their final memory position. The text segment is then written to flash ROM, at the location that was previously allocated. The memory allocated for the data and BSS segments are used as an intermediate storage for transferring text segment data from the ELF/CELF file before it is written to flash ROM. Finally, the memory allocated for the BSS segment is cleared, and the contents of the data segment is copied from the ELF/CELF file.

5.3.3 Executing the Loaded Program

When the dynamic linker has successfully loaded the code and data segments, Contiki starts executing the program.

The loaded program may replace an already running Contiki service. If the service that is to be replaced needs to pass state to the newly loaded service, Contiki supports the allocation of an external memory buffer for this purpose. However, experience has shown that this mechanism has been very scarcely used in practice and the mechanism is likely to be removed in future versions of Contiki.

5.3.4 Portability

Since the ELF/CELF format is the same across different platforms, we designed the Contiki dynamic linker to be easily portable to new platforms. The loader is split into one architecture specific part and one generic part. The generic part parses the ELF/CELF file, finds the relevant sections of the file, looks up symbols from the symbol table, and performs the generic relocation logic. The architecture specific part does only three things: allocates ROM and RAM, writes the linked and relocated binary to flash ROM, and understands the relocation types in order to modify machine code instructions that need adjustment because of relocation.

5.3.5 Alternative Designs

The Contiki core symbol table contains all externally visible symbols in the Contiki core. Many of the symbols may never need to be accessed by loadable programs, thus causing ROM overhead. An alternative design would be to let the symbol table include only a handful of symbols, entry points, that define the only ways for an application program to interact with the core. This would lead to a smaller symbol table, but would also require a detailed specification of which entry points that should be included in the symbol table. The main reason why we did not chose this design, however, is that we wish to be able to replace modules at any level of the system. For this reason, we chose to provide the same amount of symbols to an application program as it would have, would it have been compiled directly into the core. However, we are continuing to investigate this alternative design for future versions of the system.

5.4 The Java Virtual Machine

We ported the Java virtual machine (JVM) from lejOS [8], a small operating system originally developed for the Lego Mindstorms. The Lego Mindstorms are equipped with an Hitachi H8 microcontroller with 32 kilobytes of RAM available for user programs such as the JVM. The lejOS JVM works within this constrained memory while featuring preemptive threads, recursion, synchronization and exceptions. The Contiki port required changes to the RAM-only model of the lejOS JVM. To be able to run Java programs within the 2 kilobytes of RAM available on our hardware platform, Java classes needs to be stored in flash ROM rather than in RAM. The Contiki port stores the class descriptions including bytecode in flash ROM memory. Static class data and class flags that denote if classes have been initialized are stored in RAM

as well as object instances and execution stacks. The RAM requirements for the Java part of typical sensor applications are a few hundred bytes.

Java programs can call native code methods by declaring native Java methods. The Java virtual machine dispatches calls to native methods to native code. Any native function in Contiki may be called, including services that are part of a loaded Contiki program.

5.5 CVM - the Contiki Virtual Machine

We designed the Contiki Virtual Machine, CVS, to be a compromise between an application-specific and a generic virtual machine. CVM can be configured for the application running on top of the machine by allowing functions to be either implemented as native code or as CVM code. To be able to run the same programs for the Java VM and for CVM, we developed a compiler that compiles a subset of the Java language to CVM bytecode.

The design of CVM is intentionally similar to other virtual machines, including Maté [19], VM* [18], and the Java virtual machine. CVM is a stack-based machine with separated code and data areas. The CVM instruction set contains integer arithmetic, unconditional and conditional branches, and method invocation instructions. Method invocation can be done in two ways, either by invocation of CVM bytecode functions, or by invocation of functions implemented in native code. Invocation of native functions is done through a special instruction for calling native code. This instruction takes one parameter, which identifies the native function that is to be called. The native function identifiers are defined at compile time by the user that compiles a list of native functions that the CVM program should be able to call. With the native function interface, it is possible for a CVM program to call any native functions provided by the underlying system, including services provided by loadable programs.

Native functions in a CVM program are invoked like any other function. The CVM compiler uses the list of native functions to translate calls to such functions into the special instruction for calling native code. Parameters are passed to native functions through the CVM stack.

6 Evaluation

To evaluate dynamic linking of native code we compare the energy costs of transferring, linking, relocating, loading, and executing a native code module in ELF format using dynamic linking with the energy costs of transferring, loading, and executing the same program compiled for the CVM and the Java virtual machine. We devise a simple model of the energy consumption of the reprogramming process. Thereafter we experimentally quantify the energy and memory consumption as well as the execution overhead for the reprogramming, the execution methods and the applications. We use the results of the measurements as input into the model which enables us to perform a quantitative comparison of the energy-efficiency of the reprogramming methods.

We use the ESB board [33] and the Telos Sky board [29] as our experimental platforms. The ESB is equipped with an MSP430 microcontroller with 2 kilobytes of RAM and 60 kilobytes of flash ROM, an external 64 kilobyte EEPROM, as well as a set of sensors and a TR1001 radio transceiver.

```
PROCESS_THREAD(test_blink, ev, data)
{
  static struct etimer t;
  PROCESS_BEGIN();

  etimer_set(&t, CLOCK_SECOND);

  while(1) {
    leds_on(LEDS_GREEN);
    PROCESS_WAIT_UNTIL(etimer_expired(&t));
    etimer_reset(&t);

    leds_off(LEDS_GREEN);
    PROCESS_WAIT_UNTIL(etimer_expired(&t));
    etimer_reset(&t);
  }

  PROCESS_END();
}
```

Figure 3. Example Contiki program that toggles the LEDs every second.

The Telos Sky is equipped with an MSP430 microcontroller with 10 kilobytes of RAM and 48 kilobytes of flash ROM together with a CC2420 radio transceiver. We use the ESB to measure the energy of receiving, storing, linking, relocating, loading and executing loadable modules and the Telos Sky to measure the energy of receiving loadable modules.

We use three Contiki programs to measure the energy efficiency and execution overhead of our different approaches. Blinker, the first of the two programs, is shown in Figure 3. It is a simple program that toggles the LEDs every second. The second program, Object Tracker, is an object tracking application based on abstract regions [35]. To allow running the programs both as native code, as CVM code, and as Java code we have implemented these programs both in C and Java. A schematic illustration of the C implementation is in Figure 4. To support the object tracker program, we implemented a subset of the abstract regions mechanism in Contiki. The Java and CVM versions of the program call native code versions of the abstract regions functions. The third program is a simple 8 by 8 vector convolution calculation.

6.1 Energy Consumption

We model the energy consumption E of the reprogramming process with

$$E = E_p + E_s + E_l + E_f$$

where E_p is the energy spent in transferring the object over the network, E_s the energy cost of storing the object on the device, E_l the energy consumed by linking and relocating the object, and E_f the required energy for of storing the linked program in flash ROM. We use a simplified model of the network propagation energy where we assume a propagation protocol where the energy consumption E_p is proportional to the size of the object to be transferred. Formally,

$$E_p = P_p s_o$$

where s_o is the size of the object file to be transfered and P_p is a constant scale factor that depends on the network protocol used to transfer the object. We use similar equations for E_s (energy for storing the binary) and E_l (energy for linking and relocating). The equation for E_f (the energy for load-

```
PROCESS_THREAD(use_regions_process, ev, data)
{
  PROCESS_BEGIN();

  while(1) {

    value = pir_sensor.value();

    region_put(reading_key, value);
    region_put(reg_x_key, value * loc_x());
    region_put(reg_y_key, value * loc_y());
    if(value > threshold) {
      max = region_max(reading_key);

      if(max == value) {
        sum = region_sum(reading_key);
        sum_x = region_sum(reg_x_key);
        sum_y = region_sum(reg_y_key);
        centroid_x = sum_x / sum;
        centroid_y = sum_y / sum;
        send(centroid_x, centroid_y);
      }
    }

    etimer_set(&t, PERIODIC_DELAY);
    PROCESS_WAIT_UNTIL(etimer_expired(&t));

  }

  PROCESS_END();
}
```

Figure 4. Schematic implementation of an object tracker based on abstract regions.

ing the binary to ROM) contains the size of the compiled code size of the program instead of the size of the object file. This model is intentionally simple and we consider it good enough for our purpose of comparing the energy-efficiency of different reprogramming schemes.

6.1.1 Lower Bounds on Radio Reception Energy

We measured the energy consumption of receiving data over the radio for two different radio transceivers: the TR1001 [32], that is used on the ESB board, and the CC2420 [2], that conforms to the IEEE 802.15.4 standard [11] and is used on the Telos Sky board. The TR1001 provides a very low-level interface to the radio medium. The transceiver decodes data at the bit level and transmits the bits in real-time to the CPU. Start bit detection, framing, MAC layer, checksums, and all protocol processing must be done in software running on the CPU. In contrast, the interface provided by the CC2420 is at a higher level. Start bits, framing, and parts of the MAC protocol are handled by the transceiver. The software driver handles incoming and outgoing data on the packet level.

Since the TR1001 operates at the bit-level, the communication speed of the TR1001 is determined by the CPU. We use a data rate of 9600 bits per second. The CC2420 has a data rate of 250 kilobits per second, but also incurs some protocol overhead as it provides a more high-level interface.

Figure 5 shows the current draw from receiving 1000 bytes of data with the TR1001 and CC2420 radio transceivers. These measurements constitute a lower bound on the energy consumption for receiving data over the radio, as they do not include any control overhead caused by a code propagation protocol. Nor do they include any packet headers. An actual propagation protocol would incur overhead

Transceiver	Time (s)	Energy (mJ)	Time per byte (s)	Energy per byte (mJ)
TR1001	0.83	21	0.0008	0.021
CC2420	0.060	4.8	0.00006	0.0048

Table 2. Lower bounds on the time and energy consumption for receiving 1000 bytes with the TR1001 and CC2420 transceivers. All values are rounded to two significant digits.

because of both packet headers and control traffic. For example, the Deluge protocol has a control packet overhead of approximately 20% [14]. This overhead is derived from the total number of control packets and the total number of data packets in a sensor network. The average overhead in terms of number of excessive data packets received is 3.35 [14]. In addition to the actual code propagation protocol overhead, there is also overhead from the MAC layer, both in terms of packet headers and control traffic.

The TR1001 provides a low-level interface to the CPU, which enabled us to measure only the current draw of the receiver. We first measured the time required for receiving one byte of data from the radio. To produce the graph in the figure, we measured the current draw of an ESB board which we had programmed to turn on receive mode and busy-wait for the time corresponding to the reception time of 1000 bytes.

When measuring the reception current draw of the CC2420, we could not measure the time required for receiving one byte because the CC2420 does not provide an interface at the bit level. Instead, we used two Telos Sky boards and programmed one to continuously send back-to-back packets with 100 bytes of data. We programmed the other board to turn on receive mode when the on-board button was pressed. The receiver would receive 1000 bytes of data, corresponding to 10 packets, before turning the receiver off. We placed the two boards next to each other on a table to avoid packet drops. We produced the graph in Figure 5 by measuring the current draw of the receiver Telos Sky board. To ensure that we did not get spurious packet drops, we repeated the measurement five times without obtaining differing results.

Table 2 shows the lower bounds on the time and energy consumption for receiving data with the TR1001 and CC2420 transceivers. The results show that while the current draw of the CC2420 is higher than that of the TR1001, the energy efficiency in terms of energy per byte of the CC2420 is better because of the shorter time required to receive the data.

6.1.2 Energy Consumption of Dynamic Linking

To evaluate the energy consumption of dynamic linking, we measure the energy required for the Contiki dynamic linker to link and load two Contiki programs. Normally, Contiki loads programs from the radio network but to avoid measuring any unrelated radio or network effects, we stored the loadable object files in flash ROM before running the experiments. The loadable objects were stored as ELF files from which all debugging information and symbols that were not needed for run-time linking was removed. At boot-up,

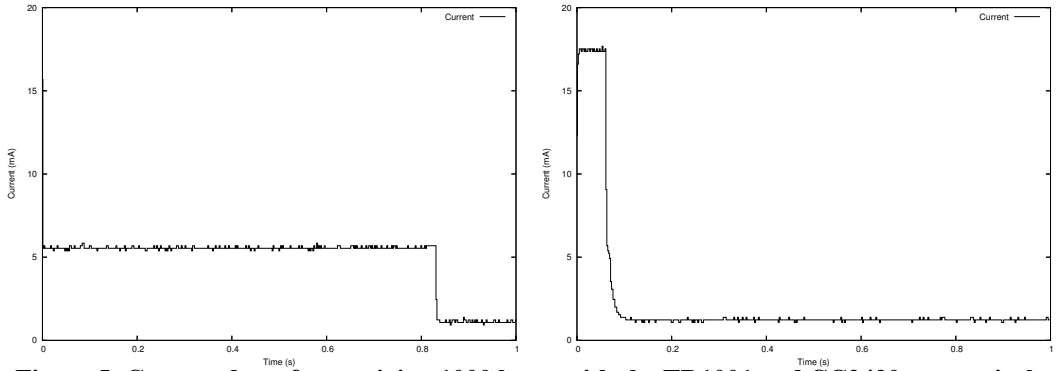

Figure 5. Current draw for receiving 1000 bytes with the TR1001 and CC2420, respectively.

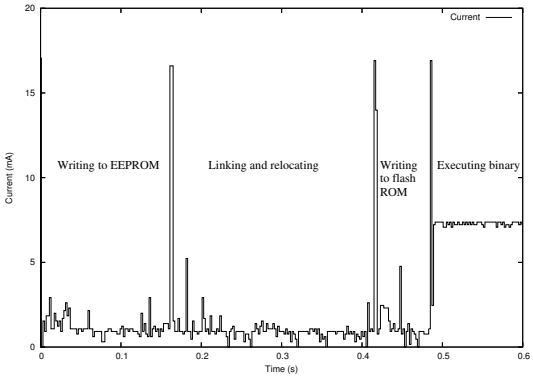

Figure 6. Current draw for writing the Blinker ELF file to EEPROM (0 - 0.166 s), linking and relocating the program (0.166 - 0.418 s), writing the resulting code to flash ROM (0.418 - 0.488 s), and executing the binary (0.488 s and onward). The current spikes delimit the three steps and are intentionally caused by blinking on-board LEDs. The high energy consumption when executing the binary is caused by the green LED.

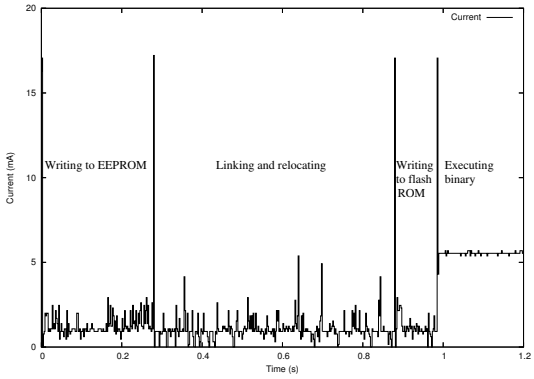

Figure 7. Current draw for writing the Object Tracker ELF file to EEPROM (0 - 0.282 s), linking and relocating the program (0.282 - 0.882 s), writing the resulting code to flash ROM (0.882 - 0.988 s), and executing the binary (0.988 s and onward). The current spikes delimit the three steps and are intentionally caused by blinking on-board LEDs. The high current draw when executing the binary comes from the radio being turned on.

one ELF file was copied into an on-board EEPROM from where the Contiki dynamic linker linked and relocated the ELF file before it loaded the program into flash ROM.

Figure 6 shows the current draw when loading the Blinker program, and Figure 7 shows the current draw when loading the Object Tracker program. The current spikes seen in both graphs are intentionally caused by blinking the on-board LEDs. The spikes delimit the four different steps that the loader is going through: copying the ELF object file to EEPROM, linking and relocating the object code, copying the linked code to flash ROM, and finally executing the loaded program. The current draw of the green LED is slightly above 8 mA, which causes the high current draw when executing the blinker program (Figure 6). Similarly, when the object tracking application starts, it turns on the radio for neighbor discovery. This causes the current draw to rise to around 6 mA in Figure 7, and matches the radio current measurements in Figure 5.

Table 3 shows the energy consumption of loading and linking the Blinker program. The energy was obtained from integration of the curve from Figure 6 and multiplying it by the voltage used in our experiments (4.5 V). We see that the linking and relocation step is the most expensive in terms of energy. It is also the longest step.

To evaluate the energy overhead of the ELF file format, we compare the energy consumption for receiving four different Contiki programs using the ELF and CELF formats. In addition to the two programs from Figures 3 and 4 we include the code for the Contiki code propagation mechanism and a network publish/subscribe program that performs periodic flooding and converging of information. The two latter programs are significantly larger. We calculate an estimate of the required energy for receiving the files by using the measured energy consumption of the CC2420 radio transceiver and multiply it by the average overhead by the Deluge code propagation protocol, 3.35 [14]. The results are listed in Table 4 and show that radio reception is more energy consuming than linking and loading a program, even for a small program. Furthermore, the results show that the relative average size and energy overhead for ELF files compared to the code and data contained in the files is approximately 4 whereas the relative CELF overhead is just under 2.

Program	Code size	Data size	ELF file size	ELF file size overhead	ELF radio reception energy (mJ)	CELF file size	CELF file size overhead	CELF radio reception energy (mJ)
Blinker	130	14	1056	7.3	17	361	2.5	5.9
Object tracker	344	22	1668	5.0	29	758	2.0	12
Code propagator	2184	10	5696	2.6	92	3686	1.7	59
Flood/converge	4298	42	8456	1.9	136	5399	1.2	87

Table 4. The overhead of the ELF and CELF file formats in terms of bytes and estimated reception energy for four Contiki programs. The reception energy is the lower bound of the radio reception energy with the CC2420 chip, multiplied by the average Deluge overhead (3.35).

Step	Blinker time (s)	Energy (mJ)	Obj. Tr. time (s)	Energy (mJ)
Wrt. EEPROM	0.164	1.1	0.282	1.9
Link & reloc	0.252	1.2	0.600	2.9
Wrt. flash ROM	0.070	0.62	0.106	0.76
Total	0.486	2.9	0.988	5.5

Table 3. Measured energy consumption of the storing, linking and loading of the 1056 bytes large Blinker binary and the 1824 bytes large Object Tracker binary. The size of the Blinker code is 130 bytes and the size of the Object Tracker code is 344 bytes.

Module	ROM	RAM
Static loader	670	0
Dynamic linker, loader	5694	18
CVM	1344	8
Java VM	13284	59

Table 5. Memory requirements, in bytes. The ROM size for the dynamic linker includes the symbol table. The RAM figures do not include memory for programs running on top of the virtual machines.

6.2 Memory Consumption

Memory consumption is an important metric for sensor nodes since memory is a scarce resource on most sensor node platforms. The ESB nodes feature only 2 KB RAM and 60 KB ROM while Mica2 motes provide 128 KB of program memory and 4 KB of RAM. The less memory required for reprogramming, the more is left for applications and support for other important tasks such as security which may also require a large part of the available memory [28].

Table 5 lists the memory requirements of the static linker, the dynamic linker and loader, the CVM and the Java VM. The dynamic linker needs to keep a table of all core symbols in the system. For a complete Contiki system with process management, networking, the dynamic loader, memory allocation, Contiki libraries, and parts of the standard C library, the symbol table requires about 4 kilobytes of ROM. This is included in the ROM size for the dynamic linker.

6.3 Execution Overhead

To measure the execution overhead of the application specific virtual machine and the Java virtual machine, we implemented the object tracking program in Figure 4 in C and Java. We compiled the Java code to CVM code and Java bytecode. We ran the compiled code on the MSP430-equipped ESB board. The native C code was compiled with the MSP430 port of GCC version 3.2.3. The MSP430 digitally-controlled oscillator was set to clock the CPU at a speed of 2.4576 MHz. We measured the execution time of the three implementations using the on-chip timer A1 that was set to generate a timer interrupt 1000 times per second. The execution times are averaged over 5000 iterations of the object tracking program.

Execution type	Execution time (ms)	Energy (mJ)
Native	0.479	0.00054
CVM	0.845	0.00095
Java VM	1.79	0.0020

Table 6. Execution times and energy consumption of one iteration of the tracking program.

Execution type	Execution time (ms)	Energy (mJ)
Native	0.67	0.00075
CVM	58.52	0.065
Java VM	65.6	0.073

Table 7. Execution times and energy consumption of the 8 by 8 vector convolution.

The results in Table 6 show the execution time of one run of the object tracking application from Figure 4. The execution time measurements are averaged over 5000 runs of the object tracking program. The energy consumption is calculated by multiplying the execution time with the average energy consumption when a program is running with the radio turned off. The table shows that the overhead of the Java virtual machine is higher than that of the CVM, which is turn is higher than the execution overhead of the native C code.

All three implementations of the tracker program use the same abstract regions library which is compiled as native code. Thus much of the execution time in the Java VM and CVM implementations of the object tracking program is spent executing the native code in the abstract regions library. Essentially, the virtual machine simply acts as a dispatcher of calls to various native functions. For programs that spend a significant part of their time executing virtual machine code the relative execution times are significantly higher for the virtual machine programs. To illustrate this, Table 7 lists the execution times of a convolution operation of two vectors of length 8. Convolution is a common operation in digital signal processing where it is used for algorithms such as filtering or edge detection. We see that the execution time of the program running on the virtual machines is close to ten times that of the native program.

Step	Dynamic linking (mJ)	Full image replacement (mJ)
Receiving	17	330
Wrt. EEPROM	1.1	22
Link & reloc	1.4	-
Wrt. flash ROM	0.45	72
Total	20	424

Table 8. Comparison of energy-consumption of reprogramming the blinker application using dynamic linking with an ELF file and full image replacement methods.

Step	ELF	CELF	CVM	Java
Size (bytes)	1824	968	123	1356
Receiving	29	12	2.0	22
Wrt. EEPROM	1.9	0.80	-	-
Link & reloc	2.5	2.5	-	-
Wrt. flash ROM	1.2	1.2	-	4.7
Total	35	16.5	2.0	26.7

Table 9. Comparison of energy-consumption in mJ of reprogramming for the object tracking application using the four different methods.

6.4 Quantitative Comparison

Using our model from Section 6.1 and the results from the above measurements, we can calculate approximations of the energy consumption for distribution, reprogramming, and execution of native and virtual machine programs in order to compare the methods with each other. We set P_p, the scale factor of the energy consumption for receiving an object file, to the average Deluge overhead of 3.35.

6.4.1 Dynamic Linking vs Full Image Replacement

We first compare the energy costs for the two native code reprogramming models: dynamic linking and full image replacement. Table 8 shows the results for the energy consumption of reprogramming the blinker application. The size of blinker application including the operating system is 20 KB which is about 20 times the size of the blinker application itself. Even though no linking needs to be performed during the full image replacement, this method is about 20 times more expensive to perform a whole image replacement compared to a modular update using the dynamic linker.

6.4.2 Dynamic Linking vs Virtual Machines

We use the tracking application to compare reprogramming using the Contiki dynamic linker with code updates for the CVM and the Java virtual machine. CVM programs are typically very small and are not stored in EEPROM, nor are they linked or written to flash. Java uncompressed class files are loaded into flash ROM before they are executed. Table 9 shows the sizes of the corresponding binaries and the energy consumption of each reprogramming step.

As expected, the process of updating sensor nodes with native code is less energy-efficient than updating with a virtual machine. Also, as shown in Table 6, executing native code is more energy-efficient than executing code for the virtual machines.

By combining the results in Table 6 and Table 9, we can compute break-even points for how often we can execute native code as opposed to virtual machine code for the same energy consumption. That is, after how many program iterations do the cheaper execution costs outweigh the more expensive code updates.

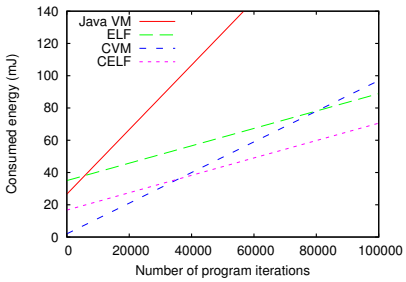

Figure 8. Break-even points for the object tracking program implemented with four different linking and execution methods.

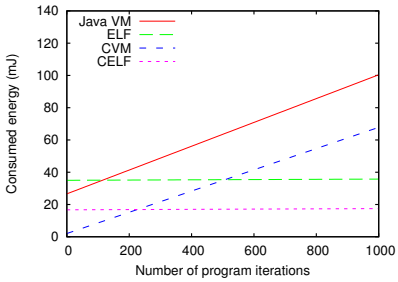

Figure 9. Break-even points for the vector convolution implemented with four different linking and execution methods.

Figure 8 shows the modeled energy consumption for executing the Object Tracking program using native code loaded with an ELF object file, native code loaded with an CELF object file, CVM code, and Java code. We see that the Java virtual machine is expensive in terms of energy and will always require more energy than native code loaded with a CELF file. For native code loaded with an ELF file the energy overhead due to receiving the file makes the Java virtual machine more energy efficient until the program is repeated a few thousand times. Due to the small size of the CVM code it is very energy efficient for small numbers of program iterations. It takes about 40000 iterations of the program before the interpretation overhead outweigh the linking and loading overhead of same program running as native code and loaded as a CELF file. If the native program was loaded with an ELF file, however, the CVM program needs to be run approximately 80000 iterations before the energy costs are the same. At the break-even point, the energy consumption is only about one fifth of the energy consumption for loading the blink program using full image replacement as shown in Table 8.

In contrast with Figure 8, Figure 9 contains the break-even points from the vector convolution in Table 7. We assume that the convolution algorithm is part of a program with the same size as in Figure 8 so that the energy consumption for reprogramming is the same. In this case the break-even points are drastically lower than in Figure 8. Here the native code loaded with an ELF file outperforms the Java imple-

mentation already at 100 iterations. The CVM implementation has spent as much energy as the native ELF implementation after 500 iterations.

6.5 Scenario Suitability

We can now apply our results to the software update scenarios discussed in Section 2. In a scenario with frequent code updates, such as the dynamic application scenario or during software development, a low loading overhead is to prefer. From Figure 8 we see that both an application-specific virtual machine and a Java machine may be good choices. Depending on the type of application it may be beneficial to decide to run the program on top of a more flexible virtual machine such as the Java machine. The price for such a decision is higher energy overhead.

In scenarios where the update frequency is low, e.g. when fixing bugs in installed software or when reconfiguring an installed application, the higher price for dynamic linking may be worth paying. If the program is continuously run for a long time, the energy savings of being able to use native code outweigh the energy cost of the linking process. Furthermore, with a virtual machine it may not be possible to make changes to all levels of the system. For example, a bug in a low-level driver can usually only be fixed by installing new native code. Moreover, programs that are computationally heavy benefit from being implemented as native code as native code has lower energy consumption than virtual machine code.

The results from Figures 8 and 9 suggest that a combination of virtual machin code and native code can be energy efficient. For many situations this may be a viable alternative to running only native code or only virtual machine code.

6.6 Portability

Because of the diversity of sensor network platforms, the Contiki dynamic linker is designed to be portable between different microcontrollers. The dynamic linker is divided into two modules: a generic part that parses and analyzes the ELF/CELF that is to be loaded, and a microcontroller-specific part that allocates memory for the program to be loaded, performs code and data relocation, and writes the linked program into memory.

To evaluate the portability of our design we have ported the dynamic linker to two different microcontrollers: the TI MSP430 and the Atmel AVR. The TI MSP430 is used in several sensor network platforms, including the Telos Sky and the ESB. The Atmel AVR is used in the Mica2 motes.

Table 10 shows the number of lines of code needed to implement each module. The dramatic difference between the MSP430-specific module and the AVR-specific module is due to the different addressing modes used by the machine code of the two microcontrollers. While the MSP430 has only one addressing mode, the AVR has 19 different addressing modes. Each addressing mode must be handled differently by the relocation function, which leads to a larger amount of code for the AVR-specific module.

Module	Lines of code, total	Lines of code, relocation function
Generic linker	292	
MSP430-specific	45	8
AVR-specific	143	104

Table 10. Number of lines of code for the dynamic linker and the microcontroller-specific parts.

7 Discussion

Standard file formats. Our main motivation behind choosing the ELF format for dynamic linking in Contiki was that the ELF format is a standard file format. Many compilers and utilities, including all GCC utilities, are able to produce and handle ELF files. Hence no special software is needed to compile and upload new programs into a network of Contiki nodes. In contrast, FlexCup [27] or diff-based approaches require the usage of specially crafted utilities to produce meta data or diff scripts required for uploading software. These special utilities also need to be maintained and ported to the full range of development platforms used for software development for the system.

Operating system support. Dynamic linking of ELF files requires support from the underlying operating system and cannot be done on monolithic operating systems such as TinyOS. This is a disadvantage of our approach. For monolithic operating systems, an approach such as FlexCup is better suited.

Heterogeneity. With diff-based approaches a binary diff is created either at a base station or by an outside server. The server must have knowledge of the exact software configuration of the sensor nodes on which the diff script is to be run. If sensor nodes are running different versions of their software, diff-based approaches do not scale.

Specifically, in many of our development networks we have witnessed a form of *micro heterogeneity* in the software configuration. Many sensor nodes, which have been running the exact same version of the Contiki operating system, have had small differences in the address of functions and variables in the core. This micro heterogeneity comes from the different core images being compiled by different developers, each having slightly different versions of the C compiler, the C library and the linker utilities. This results in small variations of the operating system image depending on which developer compiled the operating system image. With diff-based approaches micro heterogeneity poses a big problem, as the base station would have to be aware of all the small differences between each node.

Combination of native and virtual machine code. Our results suggest that a combination of native and virtual machine code is an energy efficient alternative to pure native code or pure virtual machine code approaches. The dynamic linking mechanism can be used to load the native code that is used by the virtual machine code by the native code interfaces in the virtual machines.

8 Related Work

Because of the importance of dynamic reprogramming of wireless sensor networks there has been a lot of effort in the area of software updates for sensor nodes both in the form of system support for software updates and execution environments that directly impact the type and size of updates as well as distribution protocols for software updates.

Mainwaring et al. [26] also identified the trade-off between using virtual machine code that is more expensive to run but enables more energy-efficient updates and running native code that executes more efficiently but requires more costly updates. This trade-off has been further discussed by Levis and Culler [19] who implemented the Maté virtual machine designed to both simplify programming and to leverage energy-efficient large-scale software updates in sensor networks. Maté is implemented on top of TinyOS.

Levis and Culler later enhanced Maté by application specific virtual machines (ASVMs) [20]. They address the main limitations of Maté: flexibility, concurrency and propagation. Whereas Maté was designed for a single application domain only, ASVM supports a wide range of application domains. Further, instead of relying on broadcasts for code propagation as Maté, ASVM uses the trickle algorithm [21].

The MagnetOS [23] system uses the Java virtual machine to distribute applications across an ad hoc network of laptops. In MagnetOS, Java applications are partitioned into distributed components. The components transparently communicate by raising events. Unlike Maté and Contiki, MagnetOS targets larger platforms than sensor nodes such as PocketPC devices. SensorWare [1] is another script-based proposal for programming nodes that targets larger platforms. VM* is a framework for runtime environments for sensor networks [18]. Using this framework Koshy and Pandey have implemented a subset of the Java Virtual Machine that enables programmers to write applications in Java, and access sensing devices and I/O through native interfaces.

Mobile agent-based approaches extend the notion of injected scripts by deploying dynamic, localized and intelligent mobile agents. Using mobile agents, Fok et al. have built the Agilla platform that enables continuous reprogramming by injecting new agents into the network [9].

TinyOS uses a special description language for composing a system of smaller components [10] which are statically linked with the kernel to a complete image of the system. After linking, modifying the system is not possible [19] and hence TinyOS requires the whole image to be updated even for small code changes.

Systems that offer loadable modules besides Contiki include SOS [12] and Impala [24]. Impala features an application updater that enables software updates to be performed by linking in updated modules. Updates in Impala are coarse-grained since cross-references between different modules are not possible. Also, the software updater in Impala was only implemented for much more resource-rich hardware than our target devices. The design of SOS [12] is very similar to the Contiki system: SOS consists of a small kernel and dynamically-loaded modules. However, SOS uses position independent code to achieve relocation and jump tables for application programs to access the operating system kernel. Application programs can register function pointers with the operating system for performing inter-process communication. Position independent code is not available for all platforms, however, which limits the applicability of this approach.

FlexCup [27] enables run-time installation of software components in TinyOS and thus solves the problem that a full image replacement is required for reprogramming TinyOS applications. In contrast to our ELF-based solution, FlexCup uses a non-standard format and is less portable. Further, FlexCup requires a reboot after a program has been installed, requiring an external mechanism to save and restore the state of all other applications as well as the state of running network protocols across the reboot. Contiki does not need to be rebooted after a program has been installed.

FlexCup also requires a complete duplicate image of the binary image of the system to be stored in external flash ROM. The copy of the system image is used for constructing a new system image when a new program has been loaded. In contrast, the Contiki dynamic linker does not alter the core image when programs are loaded and therefore no external copy of the core image is needed.

Since the energy consumption of distributing code in sensor networks increases with the size of the code to be distributed several attempts have been made to reduce the size of the code to be distributed. Reijers and Langendoen [31] produce an edit script based on the difference between the modified and original executable. After various optimizations including architecture-dependent ones, the script is distributed. A similar approach has been developed by Jeong and Culler [15] who use the rsync algorithm to generate the difference between modified and original executable. Koshy and Pandey's diff-based approach [17] reduces the amount of flash rewriting by modifying the linking procedure so that functions that are not changed are not shifted.

XNP [16] was the previous default reprogramming mechanism in TinyOS which is used by the multi-hop reprogramming scheme MOAP (Multihop Over-the-Air Programming) developed to distribute node images in the sensor network. MOAP distributes data to a selective number of nodes on a neighbourhood-by-neighbourhood basis that avoids flooding [34]. In Trickle [21] virtual machine code is distributed to a network of nodes. While Trickle is restricted to single packet dissemination, Deluge adds support for the dissemination of large data objects [14].

9 Conclusions

We have presented a highly portable dynamic linker and loader that uses the standard ELF file format and compared the energy-efficiency of run-time dynamic linking with an application specific virtual machine and a Java virtual machine. We show that dynamic linking is feasible even for constrained sensor nodes.

Our results also suggest that a combination of native and virtual machine code provide an energy efficient alternative to pure native code or pure virtual machine approaches. The native code that is called from the virtual machine code can be updated using the dynamic linker, even in heterogeneous systems.

Acknowledgments

This work was partly financed by VINNOVA, the Swedish Agency for Innovation Systems, and the European Commission under contract IST-004536-RUNES. Thanks to our paper shepherd Feng Zhao for reading and commenting on the paper.

10 References

[1] A. Boulis, C. Han, and M. B. Srivastava. Design and implementation of a framework for efficient and programmable sensor networks. In *Proceedings of The First International Conference on Mobile Systems, Applications, and Services (MOBISYS '03)*, May 2003.

[2] Chipcon AS. CC2420 Datasheet (rev. 1.3), 2005. http://www.chipcon.com/

[3] TIS Committee. Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2, May 1995.

[4] H. Dai, M. Neufeld, and R. Han. Elf: an efficient log-structured flash file system for micro sensor nodes. In *SenSys*, pages 176–187, 2004.

[5] A. Dunkels, B. Grönvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the First IEEE Workshop on Embedded Networked Sensors*, Tampa, Florida, USA, November 2004.

[6] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali. Protothreads: Simplifying event-driven programming of memory-constrained embedded systems. In *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems, SenSys 2006*, Boulder, Colorado, USA, 2006.

[7] D. Estrin (editor). *Embedded everywhere: A research agenda for networked systems of embedded computers*. National Academy Press, 1st edition, October 2001. ISBN: 0309075688

[8] G. Ferrari, J. Stuber, A. Gombos, and D. Laverde, editors. *Programming Lego Mindstorms with Java with CD-ROM*. Syngress Publishing, 2002. ISBN: 1928994555

[9] C. Fok, G. Roman, and C. Lu. Rapid development and flexible deployment of adaptive wireless sensor network applications. In *Proceedings of the 24th International Conference on Distributed Computing Systems*, June 2005.

[10] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 1–11, 2003.

[11] J. A. Gutierrez, M. Naeve, E. Callaway, M. Bourgeois, V. Mitter, and B. Heile. IEEE 802.15.4: A developing standard for low-power low-cost wireless personal area networks. *IEEE Network*, 15(5):12–19, September/October 2001.

[12] C. Han, R. K. Rengaswamy, R. Shea, E. Kohler, and M. Srivastava. Sos: A dynamic operating system for sensor networks. In *MobiSYS '05: Proceedings of the 3rd international conference on Mobile systems, applications, and services*. ACM Press, 2005.

[13] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.

[14] J. W. Hui and D. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *Proc. SenSys'04*, Baltimore, Maryland, USA, November 2004.

[15] J. Jeong and D. Culler. Incremental network programming for wireless sensors. In *Proceedings of the First IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks IEEE SECON (2004)*, October 2004.

[16] J. Jeong, S. Kim, and A. Broad. Network reprogramming. TinyOS documentation, 2003. Visited 2006-04-06. http://www.tinyos.net/tinyos-1.x/doc/NetworkReprogramming.pdf

[17] J. Koshy and R. Pandey. Remote incremental linking for energy-efficient reprogramming of sensor networks. In *Proceedings of the second European Workshop on Wireless Sensor Networks*, 2005.

[18] J. Koshy and R. Pandey. Vm*: Synthesizing scalable runtime environments for sensor networks. In *Proc. SenSys'05*, San Diego, CA, USA, November 2005.

[19] P. Levis and D. Culler. Maté: A tiny virtual machine for sensor networks. In *Proceedings of ASPLOS-X*, San Jose, CA, USA, October 2002.

[20] P. Levis, D. Gay, and D Culler. Active sensor networks. In *Proc. USENIX/ACM NSDI'05*, Boston, MA, USA, May 2005.

[21] P. Levis, N. Patel, D. Culler, and S. Shenker. Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *Proc. NSDI'04*, March 2004.

[22] J. Lilius and I. Paltor. Deeply embedded python, a virtual machine for embedded systems. Web page. 2006-04-06. http://www.tucs.fi/magazin/output.php?ID=2000.N2.LilDeEmPy

[23] H. Liu, T. Roeder, K. Walsh, R. Barr, and E. Gün Sirer. Design and implementation of a single system image operating system for ad hoc networks. In *MobiSys*, pages 149–162, 2005.

[24] T. Liu, C. Sadler, P. Zhang, and M. Martonosi. Implementing software on resource-constrained mobile sensors: Experiences with Impala and ZebraNet. In *Proc. Second Intl. Conference on Mobile Systems, Applications and Services (MOBISYS 2004)*, June 2004.

[25] G. Mainland, L. Kang, S. Lahaie, D. C. Parkes, and M. Welsh. Using virtual markets to program global behavior in sensor networks. In *Proceedings of the 2004 SIGOPS European Workshop*, Leuven, Belgium, September 2004.

[26] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, and J. Anderson. Wireless sensor networks for habitat monitoring. In *First ACM Workshop on Wireless Sensor Networks and Applications (WSNA 2002)*, Atlanta, GA, USA, September 2002.

[27] P. José Marrón, M. Gauger, A. Lachenmann, D. Minder, O. Saukh, and K. Rothermel. Flexcup: A flexible and efficient code update mechanism for sensor networks. In *European Workshop on Wireless Sensor Networks*, 2006.

[28] A. Perrig, R. Szewczyk, V. Wen, D. E. Culler, and J. D. Tygar. SPINS: security protocols for sensor netowrks. In *Mobile Computing and Networking*, pages 189–199, 2001.

[29] J. Polastre, R. Szewczyk, and D. Culler. Telos: Enabling ultra-low power wireless research. In *Proc. IPSN/SPOTS'05*, Los Angeles, CA, USA, April 2005.

[30] N. Ramanathan, E. Kohler, and D. Estrin. Towards a debugging system for sensor networks. *International Journal for Network Management*, 3(5), 2005.

[31] N. Reijers and K. Langendoen. Efficient code distribution in wireless sensor networks. In *Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications*, pages 60–67, 2003.

[32] RF Monolithics. 868.35 MHz Hybrid Transceiver TR1001, 1999. http://www.rfm.com

[33] J. Schiller, H. Ritter, A. Liers, and T. Voigt. Scatterweb - low power nodes and energy aware routing. In *Proceedings of Hawaii International Conference on System Sciences*, Hawaii, USA, 2005.

[34] T. Stathopoulos, J. Heidemann, and D. Estrin. A remote code update mechanism for wireless sensor networks. Technical Report CENS-TR-30, University of California, Los Angeles, Center for Embedded Networked Computing, November 2003.

[35] M. Welsh and G. Mainland. Programming sensor networks using abstract regions. In *Proc. USENIX/ACM NSDI'04*, San Francisco, CA,, March 2004.

[36] G. Werner-Allen, P. Swieskowski, and M. Welsh. Motelab: A wireless sensor network testbed. In *Proc. IPSN/SPOTS'05*, Los Angeles, CA, USA, April 2005.