An Embedded Systems Programming Environment for C[⋆]

Bernd Burgstaller[1], Bernhard Scholz[1], and Anton Ertl[2]

[1] The University of Sydney
[2] Technische Universität Wien

Abstract. Resource constraints are a major concern with the design, development, and deployment of embedded systems. Embedded systems are highly hardware-dependent and have little computational power. Mobile embedded systems are further constrained by their limited battery capacity. Many of these systems are still programmed in assembly language because there is a lack of efficient programming environments.
To overcome or at least alleviate the restrictions, we propose a light-weight and versatile programming environment for the C programming language that offers mixed-mode execution, i.e., code is either executed on the CPU or on a virtual machine (VM). This mixed-mode execution environment combines the advantages of highly compressed bytecode with the speed of machine code.
We have implemented the programming environment and conducted experiments for selected programs of the MiBench suite and the Spec 2000. The VM has a footprint of 12 KB on the Intel IA32. Initial results show that the performance of the virtual machine is typically only 2 to 36 times slower than the binary execution, with compressed code occupying only 36%–57% of the machine code size. Combining sequences of VM instructions into new VM instructions (superinstructions) increases the execution speed and reduces the VM code size. Preliminary experiments indicate a speedup by a factor of 3.

1 Introduction

Mobile devices powered by batteries constitute a major share of today's embedded systems market. Mobile devices have embedded intelligence, which needs to be programmed. Due to the limitations in terms of power consumption, memory size, and computational power, programming mobile devices is still a difficult problem. To overcome or at least alleviate the problem of programming embedded systems, we introduce a programming environment for C. The C programming language is still the language of choice for mobile and embedded systems, with more than 78% of all surveyed embedded systems firmware and application developers employing it [1].

[⋆] This project has been supported by the ARC Discovery Project Grant "Compilation Techniques for Embedded Systems" under Contract DP 0560190 and the ARC Discovery Project Grant "Distributed Data Processing for Wireless Sensor Networks" under Contract DP 0664782.

Our programming environment provides a seamless integration of VM and machine code execution as outlined in Fig. 1. The program is stored as an image which contains bytecode[3] and machine code. Depending on whether the code is bytecode or machine code, it is executed on the VM or on the CPU respectively. Both, CPU and VM, share the same memory and the thread of execution can either jump from the VM to the machine code realm or vice versa.

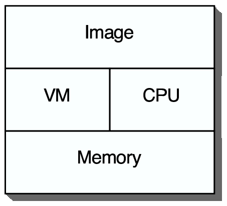

Fig. 1. Model

In this model rarely executed code is run on the VM. Frequently executed code is run on the CPU. This mixed-mode execution combines the advantage of both worlds: machine code is fast however has limited compression potential. Bytecode is stored highly compressed though the execution is slower. This execution model results in small image sizes, which reduces memory footprint and therefore devices will save energy. Also, the costs per device will decrease. Further advantages of VMs are hardware independent execution of C-programs and the fast deployment of programs by downloading them via an inter-network communication.

The contribution of this paper is the implementation of a light-weight programming environment for the C programming language. This programming environment offers mixed-mode execution, i.e., a seamless integration of VM code and machine code. The footprint overhead of the VM is small. The current footprint of the VM on an Intel IA32 architecture is 12 KB.

The paper is organised as follows: in Sec. 2 we discuss the compilation path of the programming environment. In Sec. 3 we discuss the design of our VM. In Sec. 4 we present experimental results. In Sec. 5 we survey related work. We draw our conclusions in Sec. 6.

2 Compilation

Fig. 2 depicts the compilation path of our embedded systems programming environment. Therein an application consists of a set of C source files containing code that can be compiled to either bytecode or to machine code. To allow the programmer to select between the two, we extend the C programming language with two storage class specifiers (cf. [2]), namely `vm` and `mc`. Furthermore, we use a command line parameter with the C-compiler to select a default storage class for unassigned entities. (Unassigned entities are entities that have not been assigned one of the above storage class specifiers). With this mechanism we partition the set of entities of a given application into the set of entities assigned to the realm of the VM, and those assigned to machine code.

It is the purpose of the *splitter* to preprocess an application and separate each source file into a corresponding `vm` and `mc` file that reflects the programmer's choices with respect to compilation to `vm` or `mc` code. The splitter has to achieve a

[3] In the context of this paper the word "bytecode" does not denote Java bytecode. Instead, it denotes the instruction code format of our VM.

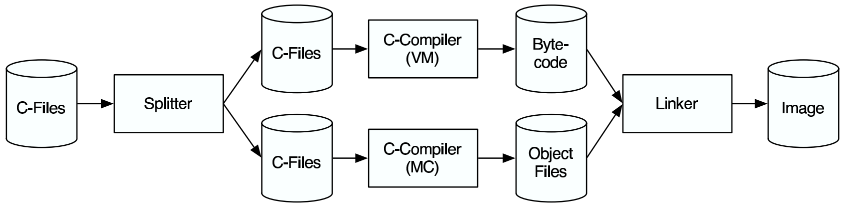

Fig. 2. Compilation Path

clear semantic separation between `vm` and `mc` code to enable separate compilation by the `vm` and `mc` compilers.

```
1   #define MAX 1024
2   mc static void fft_float (float *R_In, float *I_In, float *R_Out, float *I_Out);
3   vm char Buffer[MAX];

4   vm int main(void){
5       float R_In[MAX], I_In[MAX], R_Out[MAX], I_Out[MAX];
6       fft_float (R_In,I_In,R_Out,I_Out);
7       return 0;
8   }

9   mc static void fft_float (float *R_In, float *I_In, float *R_Out, float *I_Out)
10  { /* perform FFT */ }
```

(a) Application

```
1   #define MAX 1024
2   extern void fft_float (float *R_In, float *I_In, float *R_Out, float *I_Out);
3   char Buffer[MAX];
4   int main(void){
5       float R_In[MAX]; float I_In[MAX]; float R_Out[MAX]; float I_Out[MAX];
6       fft_float (R_In,I_In,R_Out,I_Out);
7       return 0;
8   }
```

(b) Application, `vm` Realm

```
1   #define MAX 1024
2   extern char Buffer[MAX];
3   void fft_float (float *R_In, float *I_In, float *R_Out, float *I_Out)
4   { /* perform FFT */ }
```

(c) Application, `mc` Realm

Fig. 3. Example: Splitting of Application Sources

The example in Fig. 3 is a simplified version of the FFT benchmark from the MiBench embedded benchmark suite [3]. Figure 3 (a) denotes the application which, for the sake of simplicity, consists of only one source file. Line 2 and lines 9–10 define a C function `fft_float`, which, due to the storage class specifier `mc`, is meant to be compiled to machine code. The `main` function (lines 4–8 is to be compiled to bytecode; `main` calls `fft_float`. Line 3 declares a global buffer variable that is kept in bytecode as well. Figure 3 (b) shows the `vm` file as output by the splitter. Therein the code for function `fft_float` has been removed and a corresponding external declaration has been inserted to keep the file compileable. This contrasts the declaration of the global buffer that is kept in the `vm` file (cf. line 3). All `vm` and `mc` storage class specifiers have been removed, because the occurrence of a declaration in the `vm` file already implies the `vm` storage class specifier. Likewise for the `mc`-file of Figure 3 (c). It contains only the definition of the `MAX` constant, an external declaration for the global buffer, and the code for function `fft_float`. As can be derived from Fig. 2, the separated files are then compiled by the C compilers for machine- and bytecode.

We employ LCC [4,5] for both bytecode and machine code compilation. LCC comes already equipped with a backend for bytecode, which we extended to facilitate the architecture of our VM (cf. Section 3).

Bytecode and machine code files of a given application are combined by the linker to a so-called *fat binary*. In this linkage step all references are resolved; this includes cross-references between bytecode and machine code to allow for seamless execution between the two. The fat binary can then be downloaded and executed on the embedded device.

3 Virtual Machine

The instruction set of our stack-based VM is closely related to the bytecode interface that comes with LCC [4], with the main deviations being induced by the requirements of the seamless integration of `vm` and `mc` execution. Table 1 depicts the instructions provided by our VM. The instruction opcodes cover the leftmost column whereas the column headed "IS-Op." lists operands derived from the instruction stream (all other instruction operands come from the stack). The column entitled "Suffixes" denotes the valid type suffixes for an operand (`F`=float, `I`=signed integer, `U`=unsigned integer, `P`=pointer, `V`=void, `B`=struct).[4] In this way instruction `ADDRG` receives its pointer argument `p` from the instruction stream and pushes it onto the stack. Instructions `ADDRF` and `ADDRL` receive an integer argument literal from the instruction stream; this literal is then used as an offset to the stack framepointer to compute the address of a formal or local variable. Instruction `BADDRG` uses its instruction stream argument as an index into a lookup table to derive the address of an `mc`-entity. The lookup table itself is created by the linker (cf. Section 2). For the remaining instructions of our VM we refer to the descriptions in Table 1.

To make bytecode interpretation acceptable for embedded systems, the performance of the interpretive system must be within reasonable limits compared to the performance of machine code. Due to the large design space for interpreters the achieved performance can vary drastically, with slowdowns between a factor of 10 and more than a factor of 1000 reported in the literature [6].

We used vmgen [7,8] for the implementation of our VM. Vmgen takes VM instruction descriptions as input and generates C code for execution, VM code generation, disassembly, tracing, and profiling. Vmgen already incorporates advances in interpreter technology such as threaded code (representing a VM instruction as the address of the routine that implements the instruction [9]), top

[4] Operators contain *byte size* modifiers (i.e., 1, 2, 4, 8), which we have omitted for reasons of brevity.

Instruction	IS-Op.	Suffixes	Description
`ADD SUB`	—	`FIUP..`	integer addition, subtraction
`MUL DIV`	—	`FIU...`	integer multiplication, division
`NEG`	—	`FI....`	negation
`BAND BOR BXOR`	—	`.IU...`	bitwise and, or, xor
`BCOM`	—	`.IU...`	bitwise complement
`LSH RSH MOD`	—	`.IU...`	bit shifts and remainder
`CNST`	`a`	`.IUP..`	push literal `a`
`ADDRG`	`p`	`...P..`	push address `p` of global
`ADDRF`	`l`	`...P..`	push address of formal parameter, offset `l`
`ADDRL`	`l`	`...P..`	push address of local variable, offset `l`
`BADDRG`	`index`	`...P..`	push address of `mc` entity at `index`
`INDIR`	—	`FIUP..`	pop p; push ∗p
`ASGN`	—	`FIUP..`	pop p; pop arg; ∗p = arg
`ASGN_B`	`a`	`.....B`	pop p, pop q; copy the block of length `a` at ∗q to p
`CVI`	—	`FIU...`	convert from signed integer
`CVU`	—	`.IUP..`	convert from unsigned integer
`CVF`	—	`FI....`	convert from float
`CVP`	—	`..U...`	convert from pointer
`LABEL`	—	`....V.`	label definition
`JUMP`	`target`	`....V.`	unconditional jump to `target`
`IJUMP`	—	`....V.`	indirect jump
`EQ GE GT LE LT NE`	`target`	`FIU...`	compare and jump to `target`
`ARG`	—	`FIUP..`	top of stack is next outgoing argument
`CALL`	`target`	`....V.`	`vm` procedure call to `target`
`ICALL`	—	`....V.`	pop p; call procedure at p
`INIT`	`l`	`....V.`	allocate `l` stack cells for local variables
`BCALL`	—	`FIUPVB`	`mc` procedure call
`RET`	—	`FIUPVB`	return from procedure call
`HALT`	—	`....V.`	exit the `vm` interpreter

Table 1. VM Instruction Set

of stack (TOS) caching (keeping the topmost stack element in a register), and superinstructions (combining frequently occurring patterns of VM instructions).

Figure 4 depicts a refined view of the execution architecture introduced in Sec. 1. The VM comprises a frontend, an interpreter, and stacks. The purpose of the frontend is to parse the bytecode (cf. Table 1) and to issue calls to the interpreter to build the *internal representation* that vmgen uses to store threaded instructions. Once this internal representation has been generated from the instruction stream, the interpreter is started. Our VM employs three stacks: the VM stack is used as the evaluation stack, the Arg stack holds procedure call arguments, and the Prog stack is used for machine code execution. The separate argument stack is due to LCC's ordering of bytecode instructions which intersperses procedure call arguments with other stack operands. The separate Arg stack provides an efficient way to collect procedure call arguments and arrange them in a stack frame (we will elaborate on procedure calls in the following).

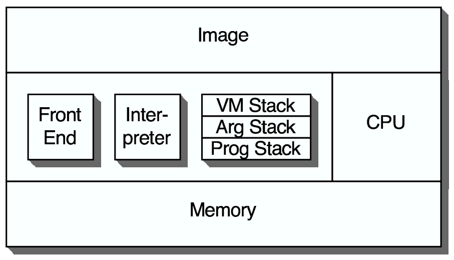

Fig. 4. Refined Execution Model

```
1 baddrg_p4 ( #ul -- p )
2 p = getsymbol_ptr(ul);
```

(a) `BADDRG_P4`

```
1 arg_p4 ( p -- ARGp )
2 /* moves p from VM
    stack to Arg stack */
```

(b) `ARG_P4`

```
1 add_i4 ( l1 l2 -- l )
2 l = l1+l2;
```

(c) `ADD_I4`

```
1 addrl_p4 (#l -- p)
2 p = (void *)(fp+l);
```

(d) `ADDRL_P4`

```
1 bcall_v ( l1 p -- )
2 indirect_call_v(p,argsp,l1);
3 argsp = argsp+l1;
```

(e) `BCALL_V`

Fig. 5. Vmgen Instruction Specifications

Vmgen provides a mechanism to specify the semantics of the instructions provided by the interpreter. As an example, consider Fig. 5 (c) which depicts the specification of the `ADD` instruction with this mechanism.[5] Therein line 1 describes the stack effect of the instruction: it pops the arguments `l1` and `l2` from the VM stack and pushes argument `l`. Line 2 contains C code that describes how argument `l` is actually computed. The overall semantics for the `ADD` instruction is to pop `l1` and `l2` from the VM stack, execute the C code, and push `l` back on the VM stack.

```
1  proc main
2  INIT 4096
3  ADDRL_P4 12288
4  ARG_P4
5  ADDRL_P4 8192
6  ARG_P4
7  ADDRL_P4 4096
8  ARG_P4
9  ADDRL_P4 0
10 ARG_P4
11 CNST_I4 4
12 CALL_V fft_float
13 CNST_I4 0
14 RET_I4
15 endproc
```

(a) main.s

```
12.1 BADDRG_P4 0
12.2 BCALL_V
```

(b) after linking

```
1 void *st[]={
2   (void *)&fft_float,
3   0L};
```

(c) address table

Fig. 6. Call from Bytecode to Machine Code

To illustrate the concept of mixed mode execution, we consider the `vm` code generated for function `main` of Fig. 3 (b). This function contains a binary call to function `fft_float`. Fig. 6 (a) depicts the bytecode for function `main` as generated by LCC. Line 2 allocates stack space on the VM stack to accommodate the four arrays of floats declared locally in `main`. The calling concept is illustrated in lines 3–12; our LCC bytecode backend is configured to evaluate parameters of function calls in the same order as with the bytecode (right to left, in this case). Each `ADDRL_P4`-instruction pushes the address of one float-array onto the

[5] Note that, unlike Table 1, the VM instructions in Fig. 5 include the type and size specifiers.

VM stack (cf. the corresponding instruction specification in Fig. 5 (d), where the address in `p` is computed relative to the VM stack framepointer `fp`). Each subsequent `ARG_P4`-instruction moves this address from the VM stack to the Arg stack (cf. Fig. 5 (b), where the `Arg`-prefix denotes the argument stack). The purpose of line 11 is to push the number of stack cells covered by the arguments onto the VM stack. Line 12 contains the call to `fft_float`.

Once our linker (cf. Fig. 2) generates the fat binary, we employ a scan of the bytecode to collect all references that cannot be resolved within the bytecode itself. For these references we generate a machine code address table. (The address table for our example is shown in Fig. 6 (c), it contains just the address of function `fft_float`.)

It is only at link time that the actual address of `fft_float` can be resolved. The linker replaces line 12 of Fig. 6 (a) by the code depicted in Fig. 6 (b) in order to account for the fact that this is a binary call. In line 12.1 the index of function `fft_float` with respect to the address table is pushed onto the VM stack (cf. Fig. 5 (a) for the corresponding instruction specification). This index is used by instruction `BCALL_V` (cf. Fig. 5 (e)) to perform the binary call.

```
void indirect_call_v(void (*f)(void),void *arg, long arglen) {
  void *p=alloca(sizeof(Cell)*arglen);
  memcpy(p,arg,sizeof(Cell)*arglen);
  return (*f)();
}
```

Fig. 7. Binary Call

The binary call mechanism itself is illustrated in Fig. 7. Function `alloca` allocates space on the program stack to account for the arguments of the call. Thereafter the arguments are copied from the Arg stack to the Prog stack (`arg` corresponds to the framepointer of the argument stack) and the binary call itself is carried out. To clean up after the call, the current argument frame is removed from the argument stack (cf. Fig. 5 (e)).

Calls of bytecode functions from machine code are carried out via a trampoline (similar to the approach in [10]). The trampoline code sets up the VM and Arg stacks and starts VM execution at the first bytecode instruction of the called function.

4 Experiments

As a testbed we used selected C programs of the MiBench benchmark suite [3] and the Spec CPU 2000 [11] benchmark suite targeting specific areas of the embedded market. We performed our experiments on the Intel IA32 platform to determine

1. the slowdown of programs executed as bytecode on our VM,
2. the VM performance improvement due to superinstructions, and
3. the best possible compression rate by using simple Huffman coding.

4.1 Performance of the Virtual Machine

We compared the performance of the VM to native code on the IA32 platform. To make a fair comparison, LCC was used to generate the bytecode and the machine code of the benchmark programs. In Table 2 the runtimes of the benchmark programs are shown.

Benchmark		Machine Code (s)	Byte Code (s)	λ
Spec2k	gzip	85	2943	34.6
	bzip2	321	11463	35.7
	mcf	55.9	483	8.6
MiBench	basicmath			
	small	0.1	0.35	3.5
	large	2.1	5.2	2.5
	bitcount	0.07	2.16	30.9
	FFT	0.16	4.3	26.9
	adpcm			
	rawcaudio	1.2	32.1	26.8
	rawdaudio	1.2	24.7	20.6
	CRC32	1.0	19.7	19.7

Table 2. Performance, Machine Code (s) $\cdot \lambda =$ Bytecode (s)

In Table 2 the time measurements are given in seconds. All programs are executed on a Pentium 4 with 1.8GHZ under Linux. The execution time of the benchmark programs vary from 0.1 to 321 seconds when compiled as machine code. If the programs are compiled as bytecode the execution increases varying from 0.35 to 11463 seconds. These results were expected since the execution of bytecode is slower than machine code. The slowdown (Column λ of Table 2) ranges from 2 to 36. Note that the slowdown increases if extensive computations are performed inside of the VM. If machine libraries are called as in basic math, the slowdown is much smaller. This result is not surprising and it goes in line with the expected slowdown factors reported in [7]. Note that this virtual machine already uses the fastest known techniques such as threaded code and an advanced dispatching mechanism, but we have not employed superinstructions in the above experiments.

Latest experiments with superinstructions enabled indicate that significant further improvements with respect to execution times are possible. In profiling the gzip program and using just the top 7% of the most frequently executed bytecode sequences we experienced a reduction of the slowdown from a factor of 36 to a factor of 12. In allowing more superinstructions further improvements can be expected. However, there is a clear tradeoff between the code size and performance of the VM. By converting 7% of the most frequently bytecode sequences the codesize of the VMs increases by 55.5%, i.e., to nearly 19 KB instead of 12 KB.

4.2 Code Compression

Bytecode has properties that allow high compression rates. In this experiment we compared the size of binary executables with Huffman encoded bytecode.

As a compression method we split the bytecode stream into three portions for Huffman coding: op-code stream, number stream, and symbol stream. This is a well known technique [12] to improve the compression rate. In this experiment we did not apply a dictionary approach (such as superinstructions or LZW) that stores re-occurring sequences only once. By adding a dictionary approach, even higher compression rates are possible. In Table 3 the results of this experiment are shown.

Benchmark		Op-Code (bits)	Number (bits)	Symbol (bits)	Total (bytes)	IR (bytes)	Object (bytes)
Spec2k	gzip	115312	46614	31194	24140	148872	42412
	bzip2	65380	33196	15816	14299	94328	28093
	mcf	26091	12417	3393	5238	39324	10325
MiBench	basicmath						
	small	4728	1965	793	936	6756	2612
	large	5970	2373	1116	1183	8648	3224
	bitcount	5702	1897	289	986	7152	1952
	FFT	6998	2926	529	1307	8756	2467
	adpcm						
	rawcaudio	2774	1309	316	550	4384	1069
	rawdaudio	2766	1310	316	549	4384	1067
	CRC32	1348	400	60	226	1772	528

Table 3. Codesize

Columns Op-Code, Number, and Symbol show the number of bits required to store the op-codes, numbers, and symbols of the bytecode. We used different Huffman codes for op-codes and arguments. Column Total gives the number of bytes to store Huffman encoded bytecode of a benchmark program. In Column Object we show the number of bytes of the Intel IA32 machine code. As shown in Fig. 8 the compression rate of compressed bytecode is very high. Here we compare the size of the compressed bytecode (Column Total) with the size of the machine code (Column Object). Compression rates range from 36% for very small programs to 57% for larger programs. Compression rates can be further improved by employing dictionary based approaches in combination with Huffman codes. This initial result is very motivating; it shows that a high compression rate is achieved by using virtual machine technology.

The number of bytes used for the internal representation is quite expensive, as shown in Column IR of Table 3. The internal representation of the bytecode is bigger than the IA32 machine code. This is attributed to the use of threaded code techniques [9] in which op-codes are replaced by function pointers. In the Intel IA32 architecture threaded code techniques waste roughly 3 bytes per bytecode instruction, i.e., four bytes for a function pointer minus one byte for an op-code. This result indicates that a buffer technique should be applied to keep most frequent executed portions of code in its internal representation. Rarely executed code should be left in its compressed form until it is needed.

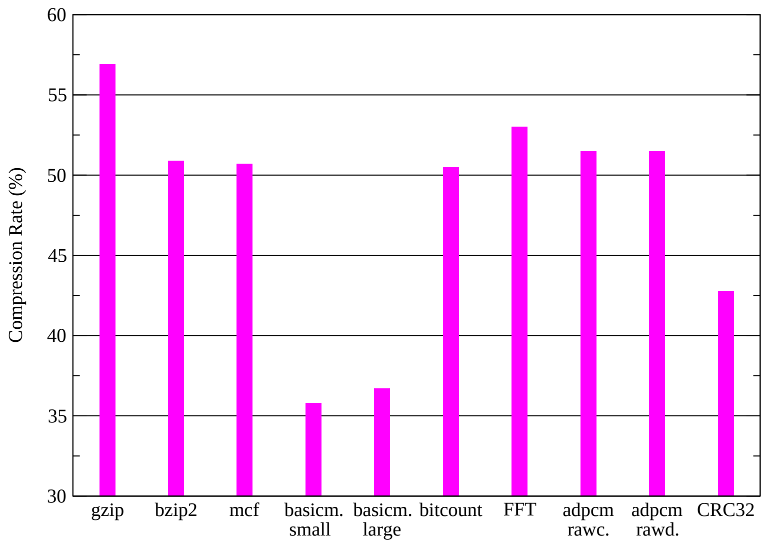

Fig. 8. Compression Rate

5 Related Work

Instead of translating the source code of a high-level language to assembly code, quite often a VM is used. A VM abstracts the properties of the underlying hardware and, therefore, makes the execution of programs hardware independent. In comparison to other implementation techniques of programming languages, VMs have the advantages of (1) portability, (2) ease of implementation, and (3) fast edit-compile-run cycles. VMs are very light-weight, which makes them suitable for embedded systems [13,14].

VM code consists of a sequence of VM instructions, which have many similarities to real machine code. In such a design, the interpretive system consists of two components: (1) a front end, that is a compiler that translates the input language to VM code, and (2) the VM interpreter that executes VM code. Good examples of such an architecture are Java's JVM [15], Prolog's WAM [16], and Smalltalk's VM [17].

Several tools [7,18,19] assist the development of VMs. A VM compiler generates an interpreter for a VM based on a VM specification. For example, the tool vmgen [7] was used to generate the code for Gforth [20].

Interpreted code can be executed with binary code and vice versa. Such a mixed execution environment was introduced for the Java programming language [21]. We believe that dynamic execution environments with mixed-mode execution have not been investigated for C, although a similar project [22] was developed for the Trimedia processor.

Low-end embedded systems have strong restrictions on the amount of available memory, which severely limits the size of the applications. Memory is a scarce commodity for several reasons: available physical space is limited, and power consumption and production costs must be minimised. Therefore, a lot of effort was taken to minimise program sizes of embedded systems applications. Especially in the realm of Java, compression rates of up to 85% of the original program size are not rare [23,24,25]. Instead of using sophisticated compression schemes, alternative representations of the VM code such as trees have been investigated [26]. For binary code several techniques have been introduced [12,27,28,29]. The main technique is to split the code into various portions and to compress them with different compression schemes. Even the instructions are split in op-codes and operands. This gives further opportunities to remove redundancies. Recently an interesting approach was introduced to incorporate compression into the instruction fetch inside a VM using Huffman codes [30]. It has to be investigated to what extend such an approach would affect the performance of our VM.

6 Conclusion

In this paper we have introduced a light-weight programming environment for the C programming language that alleviates the resource constraints present in embedded systems. Our programming environment provides seamless integration of VM and machine code execution.

In our compilation model the programmer assigns storage classes to C functions in order to decide whether they are compiled to bytecode or machine code. Our programming environment uses the LCC compiler, for which we have implemented a bytecode backend. The VM itself was developed with the `vmgen` specification tool.

We have conducted experiments with selected programs from the MiBench and Spec 2000 benchmark suites. Experiments show that the compressed bytecode occupies only 36%–57% of the corresponding machine code. The bytecode executed on the VM is only 2–36 times slower than machine code. Experiments indicate that superinstructions will further boost the performance by a factor of 3. However, superinstructions increase the footprint of the VM.

References

1. eMedia Asia Ltd. and Gartner, Inc.: Embedded Systems Development Trends: Asia. `http://www.eetasia.com` (2005)
2. Kernighan, B.W., Ritchie, D.M.: The C Programming Language. Prentice Hall Press, Upper Saddle River, NJ, USA (1988)
3. Guthaus, M.R., Ringenberg, J.S., Ernst, D., Austin, T.M., Mudge, T., Brown, R.B.: MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In: Proceedings of the IEEE 4th Annual Workshop on Workload Characterization. (2001)
4. Hanson, D.R., Fraser, C.W.: A Retargetable C Compiler: Design and Implementation. Addison Wesley (1995)
5. Fraser, C.W.: A Retargetable Compiler for ANSI C. SIGPLAN Not. **26** (1991) 29–43
6. Romer, T.H., Lee, D., Voelker, G.M., Wolman, A., Wong, W.A., Baer, J.L., Bershad, B.N., Levy, H.M.: The Structure and Performance of Interpreters. In: ASPLOS-VII: Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems, New York, NY, USA, ACM Press (1996) 150–159
7. Ertl, M.A., Gregg, D., Krall, A., Paysan, B.: `vmgen` — A Generator of Efficient Virtual Machine Interpreters. Software—Practice and Experience **32** (2002) 265–294

8. Ertl, M.A., Gregg, D.: Building an Interpreter with `vmgen`. In: Compiler Construction (CC'02), Springer LNCS 2304 (2002) 5–8 Tool Demonstration.
9. Bell, J.R.: Threaded Code. Communications of the ACM **16** (1973)
10. Bruno Haible: Foreign Function Call Libraries. `http://www.haible.de/bruno/packages-ffcall.html` (2006)
11. Standard Performance Evaluation Corporation: Spec CPU 2000 (2000) `http://www.spec.org/`.
12. Debray, S., Evans, W.: Profile-Guided Code Compression. In: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, New York, NY, USA, ACM Press (2002) 95–105
13. Levis, P., Culler, D.: Mate: A Tiny Virtual Machine for Sensor Networks. In: International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, USA. (2002)
14. Various: TinyVM () `http://tinyvm.sourceforge.net/`.
15. Lindholm, T., Yellin, F.: The Java Virtual Machine Specification. Second edn. Addison-Wesley (1999)
16. Aït-Kaci, H.: Warren's Abstract Machine: A Tutorial Reconstruction. MIT press, Cambridge (1991)
17. Goldberg, A., Robson, D.: Smalltalk-80: The Language and Its Implementation. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1983)
18. Kelsey, R.A., Rees, J.A.: A Tractable Scheme Implementation. Lisp and Symbolic Computation **7** (1994) 315–335
19. Folliot, B., Piumarta, I., Riccardi, F.: A Dynamically Configurable, Multi-Language Execution Platform. In: Proc. of the 8th ACM SIGOPS European Workshop. (1998) 175–181
20. Various: GForth () `http://www.jwdt.com/~paysan/gforth.html`.
21. Muller, G., Moura, B., Bellard, F., Consel, C.: Harissa: A Flexible and Efficient Java Environment Mixing Bytecode and Compiled Code. In: Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems, Portland, OR, USA, Usenix (1997) 1–20
22. Hoogerbrugge, J., Augusteijn, L., Trum, J., Wiel, R.V.D.: A Code Compression System Based on Pipelined Interpreters. Softw. Pract. Exper. **29** (1999) 1005–2023
23. Pugh, W.: Compressing Java Class Files. In: SIGPLAN Conference on Programming Language Design and Implementation. (1999) 247–258
24. Clausen, L.R., Schultz, U.P., Consel, C., Muller, G.: Java Bytecode Compression for Low-End Embedded Systems. ACM TOPLAS **22** (2000) 471–489
25. Bradley, Q., Horspool, R., Vitek, J.: JAZZ: An Efficient Compressed Format for Java Archive Files (1998)
26. Kistler, T., Franz, M.: A Tree-Based Alternative to Java Byte-Codes. International Journal of Parallel Programming **27** (1999) 21–33
27. Cooper, K.D., McIntosh, N.: Enhanced Code Compression for Embedded RISC Processors. In: SIGPLAN Conference on Programming Language Design and Implementation. (1999) 139–149
28. Ernst, J., Evans, W.S., Fraser, C.W., Lucco, S., Proebsting, T.A.: Code Compression. In: SIGPLAN Conference on Programming Language Design and Implementation. (1997) 358–365
29. Lekatsas, H., Wolf, W.: SAMC: A Code Compression Algorithm for Embedded Processors. IEEE Transactions on CAD **18** (1999) 1689–1701
30. Latendresse, M., Feeley, M.: Generation of Fast Interpreters for Huffman Compressed Bytecode. In: IVME '03: Proceedings of the 2003 Workshop on Interpreters, Virtual Machines and Emulators, New York, NY, USA, ACM Press (2003) 32–40