iBoy - Fast System Level Emulation Using Dynamic Binary Translation

(A Nintendo Gameboy Emulation System for Apple iPods)

Revision 2
Version History: Revision 1 on October 4, 2006

Andreas Fellnhofer, David Rigler

August 24, 2007

Abstract

Many portable devices are targeted at a mass market that justified to employ low-cost hardware components and highly performance-optimised software to reduce overall costs. Such applications often exhaustively use hardware features without any operating system layer that decouples software from architecture specific peculiarities like address space or interrupt handling. Even with the source code available it is hard to port these applications to new architectures.

Ongoing development of processors allows embedded devices to be cheap while providing high performance. This makes it feasible to use performance demanding system level emulation software that enables (binary) legacy applications to run without modifications on new platforms.

In this work we evaluate to what extent dynamic binary translation can reduce efficiency losses of system level emulation. We present an exemplary dynamic binary translator that (together with other modifications) is used to speed up the Nintendo Gameboy system level emulator gnuboy on the chosen host architecture: An Apple iPod.

1 Introduction

The Apple iPod is a portable music player that does not facilitate a dedicated chip for music decoding but two fast general purpose processors. The available performance made it feasible for the iPodLinux project to port linux to the iPod and unleash its versatile capabilities.

The fact that the iPod is a very popular portable device and its visual appearance inspired us to revive another popular portable device: the Nintendo Gameboy. Gnuboy is predestinated as a starting point for this intention as it is an open source gameboy emulator with the aim for easy portability. Therefore it was not too hard to get a first version of gnuboy running under linux on the iPod. As emulation performance was far from reaching realtime the CPU core was entirely rewritten in arm assembler.

Improvements of only 5-10% (still insufficient to reach realtime) were one of the reasons to choose a different concept for CPU emulation: dynamic binary translation. An introduction to dynamic binary translation and similar topics discussed in related papers can be found in section 2. Detailed descriptions of the 8 bit target architecture that is emulated on systemlevel and the 32 bit host architecture with its lacking memory management unit are given. Together with an introduction of gnuboys structure the constraints that outline our design are formed.

In the main part various design decisions and their alternatives are presented. The key issue is that target applications operate on a low level without utilising an operating system or system calls - hardware can be accessed in an arbitrary way and applications rely on precise timing of interrupts. A high degree of cycle accurate emulation has to be achieved to ensure correct behaviour (in terms of user perception). These considerations affect the characteristics of the dynamic translators architecture and manifest in our implementation. Aspects of the translator-system like register mapping, instruction set description, the structure of basic blocks and interrupt handling are described in detail as well as optimisations like flag liveness/constant analysis and used code caching strategies.

Finally we present the results of comprehensive evaluations that show the potential of our implementation under certain benchmark conditions.

2 Related Work

2.1 Efficient Implementation of the Smalltalk-80 System

Deutsch and Schiffman present methods to accelerate a complex programming system (Smalltalk). These methods are mainly based on the single principle of dynamic change of representation which leads to the idea to dynamically translate code for a virtual machine to native code of the executing machine. [4]

2.2 A Case For Runtime Code Generation

Keppel et al. give an introduction what runtime code generation (RTCG) is and why it has not disappeared despite of changing technology in computer systems. A system uses RTCG when binary code (especially in the form of machine instructions) is added to the instruction stream at runtime (dynamically). Two different ways to implement a dynamic compiler are described in [6]. Template based compilers use code fragments linked together depending on the instruction input stream. They often have to be written partially in assembly language of the host machine which makes them less portable. Portable compilers on the other hand use an intermediate representation which makes them more generic and flexible - the downside is higher computational effort at runtime. [6]

2.3 Shade

Cmelik and Keppel describe a fast instruction set simulator with profiling support and discusses instruction set emulation in general. In Shade basic blocks (junks of target architecture instructions) are translated to host architecture instructions at runtime forming so-called translations. Furthermore, ways to cache such translations to reduce compilation overhead are presented. The main parts of the implementation are: A compiler (or translator), a mapping of target program counter to (already compiled) translations and a virtual state (registers, memory). [3]

The compiler translates each target instruction to a series of host instructions that simulate target register and memory actions on the virtual state. A translated basic block of target instructions is called translation. To reduce the overhead of compilation, translations are stored in a cache. The target PC of the first instruction of a basic block is used to find the corresponding translation. Then the translation is executed and the target PC is set to its new value.

Basic blocks are usually rather small (a few instructions), therefore the overhead in the main loop gets dominant. To overcome this problem, translations are chained: In the predecessor the return instruction to the main loop is replaced by a branch to the entry point of the successor translation.

2.4 SimOS, Embra

Rosenblum et al. [15] present a machine simulator capable of representing processors, caches and memory systems of uni- and cachecoherent multiprocessors with choseable detail even at runtime. This allows the user to skip uninteresting parts of a program (simulating less detailed and thus faster). Witchel and Rosenblum [18] present a fast CPU simulator for SimOS called Embra that uses techniques described in [3] for dynamic binary translation but additionally handles MMU address translation and an event callback queue to support interrupts requested by SimOS. To simulate the MMU virtual addresses are mapped to physical addresses of the simulated architecture and then (via an offset) to the host architecture memory. In Embra these translations are efficiently combined in one step.

2.5 Out-of-Order Execution Techniques for BT

Bich [7] summarises binary translation techniques and extends them by an out-of-order optimisation for architectures supporting parallel execution of instructions. Exceptions occurring within reordered instructions pose a problem. Execution has to be restarted from a save point to preserve the sequential semantics.

2.6 Machine-Adaptable Dynamic Binary Translation

In [16] UQDBT, a framework for dynamic binary translation is presented which has the main goal of increasing the portability over traditional dynamic binary translators in terms of source and target architectures which is reached by using an intermediate representation of basic blocks. The task to find a specification of a generic language covering present (and possible future) architectures is non-trivial. It is pointed out that optimisations can be focused on "hot spots", parts of target code that are executed frequently.

2.7 Optimising Hot Paths in a Dynamic Binary Translator

A hot path is a sequences of native instructions (including branches) that is executed frequently so that optimisation pays off. Ung and Cifuentes demonstrate a method to find and optimise Hot-Paths. By moving and/or merging basic blocks in the hot path the locality of translated code is increased. This makes caching more efficient. Frequent executions justify strong optimisation. [17]

2.8 Dynamo

Bala et al. present a method for transparently optimising a native instruction stream before it gets executed. The key concept is to interpret the instruction stream until a hot-path is found which is a good candidate for optimisation and caching. The main optimisation of traces are redundancy removals which are applied for branches, loads and assignments. Removed branches may have side effects like changing the link register on a branch with link. Signals may be problematic since the original signals context may not be available due to optimisations (reordering of instructions, dead register elimination). Dynamo intercepts all signals. Uncritical asynchronous signals like keyboard events are queued and handled after executing the current trace. For timing critical signals the signal context is rectified. Dynamo uses a cache management that flushes the cache whenever the trace selection rate grows (indicating new hot-paths and therefore a new working set in the executed program). [1]

2.9 BinTrans

Machine-adaptable dynamic binary translation can be achieved without an intermediate representation as shown in [13]. The translator is directly created from two machine descriptions. Retargetable translators have to deal with changing numbers and different widths of registers for host and target architecture and therefore need register allocation. BinTrans uses a matching algorithm to find appropriate host instructions for a target instruction. Probst et al. point out some further important aspects of binary translation [14]. Efficient handling of condition bits is a key issue of fast binary translation systems. Computation of dead condition codes can be eliminated through simple liveness analysis. Register mappings should be static whenever possible (host architecture has enough registers). Self-modifying code needs to be identified at runtime to prevent execution of out-dated code.

3 Gameboy Structure

The Gameboy is a portable gaming console by Nintendo which was released 1989 in Japan and since then sold over 100 million times. Until now a several hundred cartridges (mainly games) are available [11]. A later version was equipped with a colour display. We will refer to the original gameboy with grayscale LCD as "target system". Physically it consists of the gameboy itself (with no operating system) and an exchangeable cartridge which may contain hardware suitable for a specific software program (game). [12] describes internals of the gameboy.

3.1 CPU

The CPU used in the target system is based on the Z80 CPU [5] and runs at 4 MHz.

The newer versions of Gameboy (Gameboy Color) support a special doublespeed mode, which can be activated dynamically during execution of a program. In this mode the CPU (and some internal timers) are clocked at 8 Mhz.

The main differences to the Z80 are: The target architecture has neither IX and IY registers nor the I and R registers with their corresponding instructions. Furthermore it has no alternative register set. Most 16 bit memory access instructions are missing and there are less 16 bit arithmetic functions. On the target architecture memory mapped I/O (MMIO) is used instead of the in/out instructions. Besides the execution time of all instructions is rounded up to a multiple of 4 cycles - therefore we count 4 cycles as one logical cycle. This is the convention when referring to "cycles" from now on.

3.1.1 Registers, Flags

The target CPU has seven 8 bit registers of which one is the accumulator (A). The other six 8 bit registers (B, C, D, E, H, L) can also be used in pairs forming three 16 bit registers (BC, DE, HL) mainly used for addressing. Further registers are the 16 bit wide program counter (PC) and stack pointer (SP). Many instructions set one or more of the four condition codes (flags): Zero, Negative, Halfcarry and Carry.

3.1.2 Interrupts

The interrupts on the target system are (ordered by priority): V-Blank, LCD Stat, Timer Interrupt, Serial Interrupt and Joypad. Each interrupt can be enabled or disabled separately using the Interrupt Enable Register (via memory mapped I/O). The Interrupt Flag Register signals whether an interrupt is pending. Interrupts can be disabled and enabled globally by CPU instructions (EI and DI). Each interrupt vector is located at a fixed memory address starting at 0x40, 8 Byte for each. Before an interrupt handler is executed, the PC is pushed onto the stack, all interrupts are disabled and the corresponding interrupt flag is reseted. The RETI instructions enables interrupts again. The HALT instruction puts the CPU into an idle state (powersaving) from which it can be waken by an interrupt.

3.2 Memory Organisation

The address bus is 16 bit wide which results in an address space of 64 KB which is universally used for RAM, ROM and MMIO.

Some areas of the address space are switchable banks increasing the addressable memory.

3.2.1 Cartridge ROM

The lower half of the address space (32 KB) is used by the cartridge which can be used in principle very flexible but in practise one of a few standard memory controllers (called MBC) is used. They divide the address space into two blocks (16 KB each) which one of is fixed mapped to the lowest bank of the ROM (bank 0) and the other one switchable (bank 1 to n-1) by write accesses to some specific cartridge addresses. Cartridges typically provide 16 to 2048 KB of ROM.

3.2.2 RAM

The target system provides 8 KB of RAM for general purpose (Working RAM or WRAM: 0xc000-0xdfff). Further 8 KB are used as video RAM (VRAM: 0x4000-0x7fff) and 160 Byte as the Object Attribute Memory (OAM: 0xfe00-0xfe9f) which is used by the graphical subsystem described in section 5.2. The higher part of the address space (HIRAM: 0xff80-0xfffe) has several specific locations used for different MMIO tasks (sound registers, LCD registers) the remaining space of HIRAM can be used as RAM. We refer to this mixed area as HIRAM.

3.2.3 Cartridge Extensions

Some cartridges have additional hardware as mentioned above. Some important of these optional functionalities are a real-time clock (RTC) and a battery backed SRAM for non-volatile data (savegames etc.).

3.3 LCD

The LCD of the target system is 160 pixels wide and 144 pixels high and capable of displaying 4 grayscales - 2 bit per pixel (The Gameboy Color is capable of displaying 32768 different colours). The vertical refresh rate is 59.73 Hz including a V-Blank interval of 10 scanlines.

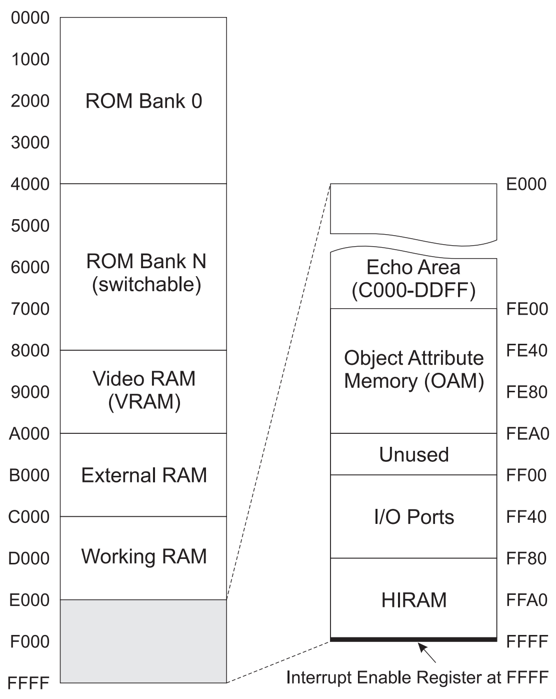

Figure 1: Memory Organisation Overview

3.3.1 VRAM Organisation

The VRAM is no linear representation (like a framebuffer) of the displayed picture but a rather complex tile oriented data structure. A visible image consists of three overlayed layers. Each layer is constructed out of tiles which are bitmaps, 8 by 8 pixels in size, consuming 16 Byte each. Each pixel has a 2 bit colour index. This all divides the VRAM into two logical parts. One for tile bitmaps (tile table) and one for the organisation of tiles.

1. **The Background Layer:** The virtual background consists of 32 by 32 tiles (256x256 pixels). Technically it is stored as an array of indices into the tile table where one tile can occur multiple times. The visible part is positioned on this background by x and y coordinates and wraps around at the borders. It can be switched on and off.

2. **The Window Layer:** It is organised similar to the background layer and overlays the background. The Window is a rectangular area reaching from a specific x and y coordinate to the bottom right corner of the visible background. It can be switched on and off.

3. **The Sprite Layer:** The OAM consists of 40 data structures each representing

one sprite. A sprite consists of one tile index (or two in special cases), a position and some attribute flags (e.g. flipping, palette number). For sprites the bitmap is interpreted in a special way, such that colour index 0 makes it transparent resulting in three usable colours for sprites. Furthermore a special flag indicates that the sprite is displayed only where the colour index of the background (window) is zero.

The target system utilises one colour palette for background (and window) and two for sprites. Each of the four indices (for object palettes three as mentioned above) can be dynamically assigned to one of the four grayscales. It should be noticed that all of the described parameters (e.g. coordinates, on/off flag) can be changed at any time especially within one LCD frame and they are applied immediately.

For the Gameboy Color the VRAM area is switchable (there are two banks 0 and 1). Bank1 is used to store Gameboy Color specific data: additional and bigger colour-palettes, new sprite attributes and mirroring parameters for all tiles.

3.4 Additional Hardware

Timer, several sound chips, keypad and the serial interface, which are not described in detail for now, complete the target system. A special DMA controller is used to fill the Object Attribute Memory from an arbitrary source address. During the DMA transfer the CPU must not access any address but HIRAM. Therefore a small procedure initialising such transfers has to be installed in HIRAM.

4 Apple iPod

4.1 Hardware

An Apple iPod of the fourth generation with grayscale display is taken as the host system. A System-On-Chip with dual ARM7TDMI microprocessors running at 75 MHz (other frequencies are possible) is used. 32 MByte of SD-RAM are shared by the two processors over two separate 8 KByte caches (no cache coherence). Additional 96 KByte of (faster) uncached SRAM make communication between the processors efficient. The display of the host architecture nearly matches the target architecture's. Its size is 160 by 128 pixels with 4 grayscales. The vertical mismatch forces target simulation to cut off 16 scanlines or downscaling. The host systems touchwheel can be used to sense the absolute position of touch points and is, like the LCD, accessed in a direct way. All other used hardware is accessed through the operating system drivers and therefore not described in detail.

4.2 Software

Because of a lacking full-featured memory management unit the GNU/Linux operating system on the host system uses a ucLinux kernel, a lightweight version of the Linux kernel specialised for systems without MMU [10]. The Gnu Compiler Collection is used to produce code for the host architecture.

5 Gnuboy Structure

Gnuboy is an open source gameboy (and gameboy colour) emulator written in C and its aim is to support a variety of host platforms. It abandons floating point operation and the usage of threads leading to a gain of portability.

5.1 CPU Emulation

The target systems CPU is emulated in a very classical way in three stages: fetch, decode and execute. All target system registers are combined to a virtual state residing in the hosts memory. Gnuboy's approach of system simulation can be roughly described as follows: First the cartridge memory is loaded into host memory and the virtual state is initialised. Then the main loop is entered, iterating several routines simulating one frame. Besides the emulation of the CPU the keypad is polled and blocks of sound are generated and written to a DSP device which is blocking and therefore generates the realtime base.

5.2 LCD Emulation

To speed up graphic emulation the tiles are cached in an array with 1 Byte per pixel. 2 Bits are used for the colour index, the others are for the palette number and to distinguish between sprite and background tiles. Whenever a write access to VRAM occurs parts of the cache are rebuilt. CPU is executing and potentially changing parameters of the LCD until a LCD Status Interrupt triggers a scanline draw. Gnuboy prefers a precise emulation strategy, therefore the unit of consideration is one scanline. One scanline is drawn in three steps as there are three graphical layers. First the tiles forming the background are copied to a buffer (from left to right) and their palette bits are set to the current background palette until the window starts or the screen ends. Then the window is handled in a similar way. In the last step all 40 sprites are evaluated if they are contained in the scanline and drawn afterwards replacing some parts of the background/window depending on their sprite attributes. Again the appropriate palette bits are set. Before displaying the scanline on the host screen the colours are resolved with their palettes with a recolour function.

5.3 Memory Access Strategies

Memory access is crucial for performance and can not be performed directly in the case of banked memory. For every access the selected bank must be taken into account to get the full (host) address. Since bank switches are far more seldom than memory accesses it pays off to rebuild a cache every time a bank is switched. Technically this cache is composed of two pointer arrays (One for read and one for write access). The high 4 bits of the address are used as an index into this array, the remaining as an offset to the stored pointer. Some memory areas can not be mapped in this way because of side affects (e.g. MMIO). For these cases zero is stored in the array and a handler function is called on read/write access.

5.4 Interrupt Handling

After each emulated instruction in the CPU loop pending interrupts are handled. An interrupt is pending if the master interrupt enable bit (IME) and the interrupt's IE and IF bits are set. To handle an interrupt, the master interrupt enable bit is reset, the current program counter is pushed to the emulated stack and set to the appropriate interrupt vector afterwards.

5.5 Further Features

Gnuboy provides a framework for configuration variables to be set during emulation. This includes colour values, keyboard mapping, sound options, scaling of the LCD and others. It is possible to save and load the emulators state.

6 Differences to Gnuboy

In contrast to Gnuboy iBoy provides an integrated GUI designed for the limited control abilities of the iPod. The biggest difference is the usage of a dynamic binary translator for instruction set simulation. This leads to some changes like different interrupt handling. The LCD core is optimised for speed rather than compatibility or accuracy. To use the host platforms potential iBoy is partitioned and executed on both processors.

6.1 LCD Core

As the main goal was to emulate the original Gameboy (grayscale) as fast as possible on grayscale iPods, we implemented a special LCD emulation module that is faster than the original method of gnuboy but does not support colours. (For emulation of Gameboy Color a slightly modified version of gnuboy original LCD engine is used.) The special grayscale LCD emulation module enables us to reduce the memory needed by the tiles cache. Each pixel is represented by 2 bits, palette bits as described before are not stored. The advantages of this format are that it is the native pixel format of the host architectures display and that 8 pixels (the width of a tile) form a 16 bit word which can be processed more efficient. This pixel format reduces memory access but needs a lot of shift operations which are cheap on the target architecture compared to memory operations [8].

The background palette is applied during the creation of the cache - the drawback of this is, that the cache has to be rebuilt every time the palette changes. It should be remembered, that changes for palettes and patterns are allowed to happen any time which may lead to excessive cache rebuilds. For that reason iBoy rebuilds the cache only once per frame, which is sufficient for most cases. As the background palette is applied to all patterns in the cache which some of may be used as sprite patterns too, the need for some compensative palette operations arises when drawing the sprites.

The sprite palette is applied on the fly when the buffer for one scanline is filled. As transparency is coded as index 0 (and shift operations ”generate” transparency) sprite pattern entries are exclusive ored with the background colour of index 0 to restore the transparency to zeros. An efficient function using bit operations applies the sprite palettes for the remaining three indices. It should be noticed that the sprite palette must be modified to correct both transformations (for background and exclusive or) each time the background or a sprite palette changes. This correction can only be done correctly (without creating artifacts) if all four colours are used in the background palette which is usually the case. Figure 2 gives an overview of pattern handling.

This all enables an efficient way to merge the sprites with the background. The background is cleared (at non transparent areas of the sprite) and a simple or operation finally merges background and sprites.

6.2 Coprocessor

Another feature to gain performance is load sharing between the two available processors on the target system. Some restrictions have to be taken into account. The two processors are not cache coherent - thus all communication has to go over the small

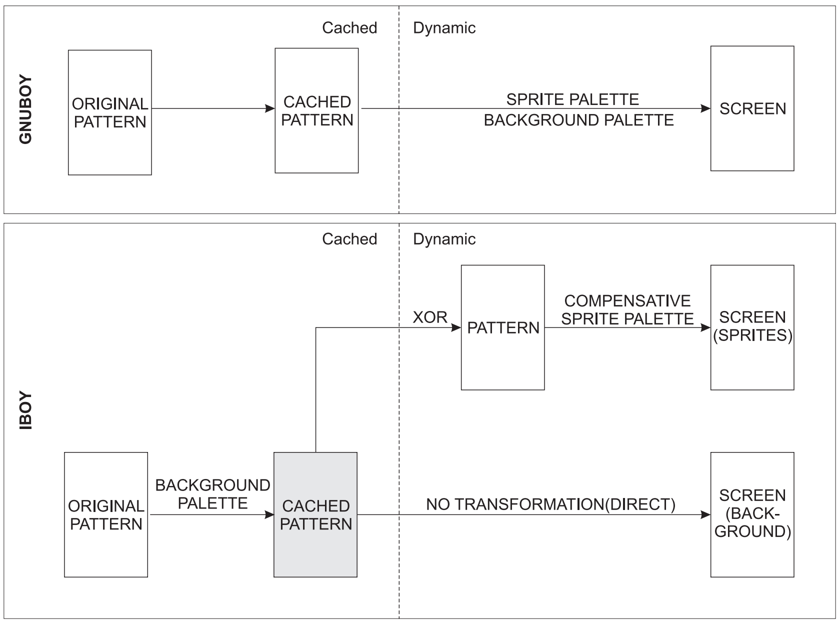

Figure 2: Comparison of Pattern Handling

SRAM which is uncached. The idea is to statically partition the simulation into two relatively independent parts which are the LCD simulation on one side and the rest on the other. This choice was made because of the nearly equal computational effort of these parts and the relatively small interface (VRAM). In fact such a partitioning was also chosen in the original target architectures hardware design.

The main processor copies the relevant HIRAM registers to the SRAM and indicates that a scanline is ready to be drawn. Then it proceeds until another scanline is ready and blocks until the coprocessor has finished with the previous scanline.

The coprocessor operates vice versa: It waits for a scanline, processes it (as described in 6.1) and blocks for the next scanline. In the VBLANK interval the main processor does not send requests for a longer period. This is used by the coprocessor to rebuild all caches if necessary.

Update: In newer versions of iBoy the main processor (emulating the CPU) does not block, regardless of the state of the second processor (emulating the LCD). This is achieved by copying the relevant HIRAM registers into an array, where each scanline has its own independent space. The second processor can rotate through those scanlines and draw them in round-robin order. If the main processor is faster this may lead to dropped scanlines (or even dropped frames), which is tolerable and does not influence user perception in most cases.

7 Dynamic Translation

7.1 Architectural Overview

When the emulator is running, the so-called cpu_emulate function is executed. The parameter for cpu_emulate determines the duration in target clock cycles the target system is to be emulated. 17555 cycles of the CPU equal the duration of one frame forming the internal timebase. Cpu_emulate is periodically executed and emulates the target system for such a duration. An internal sound buffer is filled during its execution, which is then sent to a blocking DSP device thus creating the bridge to realtime. The original frame refresh rate is 59.73 Hz. This is the goal to be reached by our emulation.

The tasks fullfilled by cpu_emulate include advancing all internal timers, performing DMA transfers and LCD transitions, handling of interrupts and simulating the target instructions on the virtual target state. Our work focuses on the latter two points. In contrast to gnuboy we use a dynamic binary translator to efficiently implement this CPU emulation.

The key idea is to dynamically (at runtime) translate chunks of target instructions into blocks of host architecture native instructions which are subsequently executed ([1], [3]). The target program counter (PC) is mapped to a host start address of translated native instructions. A lookup is performed to find already compiled blocks in the cache, if it fails the compiler is called, translating one block starting at the current target PC and caches its translation (the native block). To keep the architecture simple and elegant we decided not to do any classic emulation in parallel. The drawback is that frequently changing program regions (e.g. self modifying code in RAM areas) can lead to excessive compiling if no countermeasurements are taken. Arguments for this decision are that the vast majority of executable target code is located in ROM areas and that a simple and fast compiler design without too much (time consuming) optimisations was chosen.

After a native block is executed a new PC is provided and control is returned back to the lookup mechanism. As basic blocks are rather small (a few target instructions [3]) a further idea to gain performance is linking native blocks and thus minimising the lookup overhead. [13] Instead of returning to the lookup mechanism, control is directly passed to the next native block using backpatching (as described in section 7.7.1). It should be kept in mind that no operating system is used on the target system, target applications are at a low level and potentially depend on cycle-accurate timing of hardware interrupts and other components. This implies that the execution of native blocks has to be suspended periodically to simulate the other parts of the system, especially internal timers and interrupts and target CPU cycles need to be counted exactly.

A big part of the dynamic compiler is generated from an instruction set description (our so-called in-file).

1. The in-file (mostly host assembler) is translated to C functions capable of producing host code for a single target instruction
2. The C functions are compiled into the dynamic binary translator
3. At runtime the binary translator calls the compiled C functions producing actual host code

7.2 Register Mapping

As shown in Figure 4 the host platform has 16 registers available in the user space. 4 registers are caller-save and used as call parameters by convention. The register r12-r15 are used for special purposes like as program counter, link register or stack pointer. The remaining 8 registers are callee-save for general purpose usage.

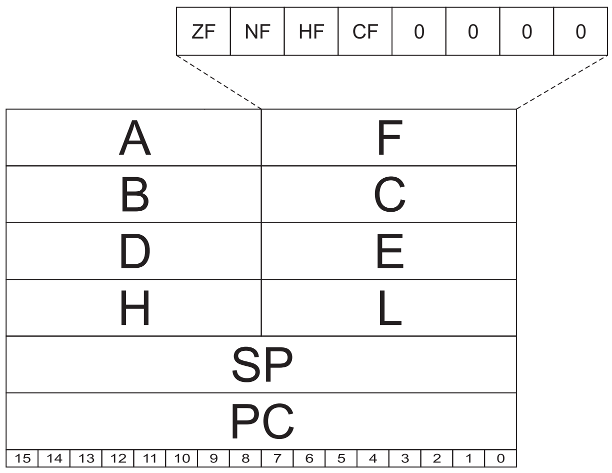

Figure 3: Target Platform Registers

The registers of the target platform are shown in Figure 3. Ten 8-Bit registers are available of which one is used as an accumulator and another for the flags. The flags are formed of 4 Bits (Zero, Negative, Half-Carry and Carry) and accessed implicit by most instructions. Only two instructions explicitly use them, namely "push af" and "pop af". Eight registers can be accessed as 16 bit pairs which allows them to be used for 16 bit operations like address calculations. Additionally a program counter and a stack pointer (each of 16 bit) are available.

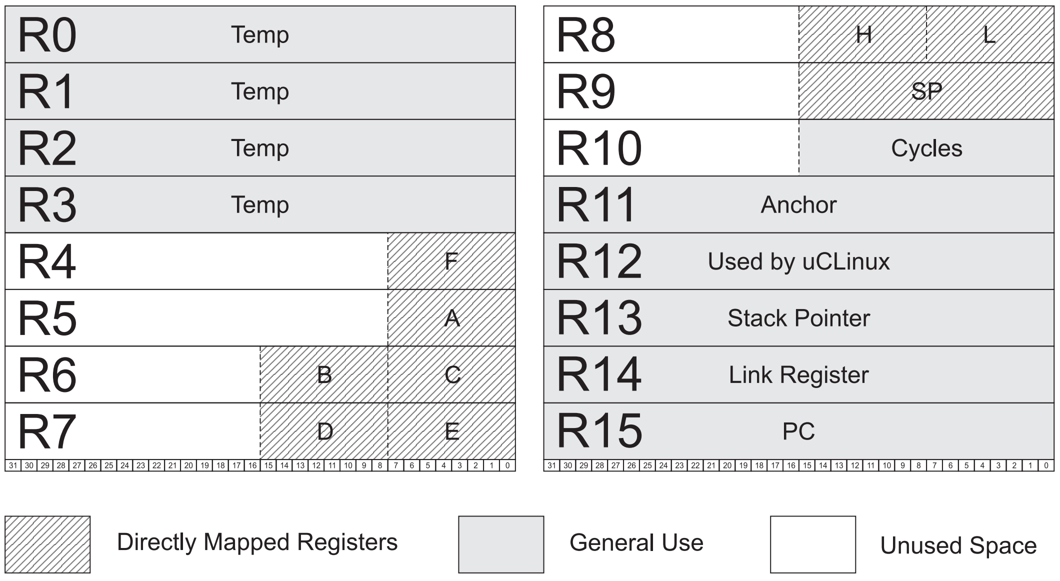

Figure 4: Host Platform Registers with Mapping

When a CPU is simulated, its registers must be stored on the host architecture. When using dynamic binary translation they are ideally placed in the host architectures registers. In our special case all target registers fit into the general purpose host registers if each 16 bit register is stored in a single 32 bit host register so that they get statically mapped. 8 bit operations using one half of the combined registers are slowed down since bit masking and shift instructions have to be performed but such instructions are cheap on arm CPUs. Accumulator and flags are stored in a single host register each. This is justified by the fact that the accumulator is the most used register and nearly always accessed in its single form (8 bit register) - in fact only two instructions (push af, pop af) use the combined AF register. A similar argument is valid for the flags (F) which are accessed bit wise. The program counter is not stored explicitly in a register as it is only relevant at compile time and is implicitly represented in dynamically compiled code.

The big advantage of such a static mapping is that it is inherently simpler than dynamic register allocation and faster than keeping target architecture registers in memory. [14]

7.3 Instruction Set Description: in-file

To translate one instruction set to another a precise description of each target instruction is a prerequisite. An instruction can be viewed as a function operating on the system state containing registers, memory, condition codes (flags) and some form of time (e.g. cycles past). Therefore the description has to include inputs and outputs of this function as well as the function itself. A similar description of the host architecture is needed if a generic (portable) approach is chosen. [13] A compiler generator has to combine host instructions in a way that matches the description of the according target instructions. A generic matching algorithm can be used for different host and target architectures - the big advantage of this rather complex approach. Complexity even grows if highly efficient translations for different architectures (or even future architectures) is aimed for.

A simpler alternative is to provide the target architectures description in the form of (hand-crafted) code templates without the need to explicitly describe the host architecture.Because of the goal to implement a thin compiler without too high complexity this approach was favoured in our work.

The actually used templates are stored in a text file. It consists of sections (one for each target instruction) with the following contents:

1. Hexadecimal representation of opcode and opcode name
2. Vector of flag read properties: "R" means flag is read, "X" flag is ignored
3. Vector of flag write properties: "0" or "1" means flag is set to a constant value, "R" means flag is written to a computed value, "X" means flag remains untouched
4. A body, beginning with "begin{" and ending with "}". It consists of the actual host assembler code that implements the specified target instruction in different variants.

Listing 1: Example section of the in-file

```
opcode(0x07)  // RLCA
// ZNHC
rf XXXX
wf 000R

begin{
        mov     r0, r5, lsr #7
        orr     r5, r0, r5, lsl #1
        if(ALIVE(FC)){
                mov     r4, r0, lsl #4
        }
        and     r5, r5, #0xff
}
```

Within templates it is possible to access the current value of the PC provided by the compiler. First, this is needed because the template itself has to read immediate operands of its opcode and second because operations explicitly use the PC (e.g. calls, relative jumps).

A major part of the computational resources is used for the calculation of flags. Flags are not always (or even seldom) used after their computation but overwritten, in other words, they are dead. This results in an opportunity for optimisation which is realised through conditional computation of flags: Constant flags are not explicitly set within a template - the generator and the compiler take care of this. Besides the compiler provides a vector of flags that are not dead and need to be computed in the template. A second vector that is provided, represents flags that values are known at compile time. A template can access these values to generate special versions of the target instruction that are potentially faster since flag values that are known at compile time do not need to be checked at runtime.

7.4 The Generator

A generator transforms the template to a C function. Assembler lines are parsed and translated to C macros representing a 32-bit ARM assembler instruction. Basically the execution of the C function fills an array provided by the compiler with one ARM instruction after another, one for each assembler line. As shown in Listing 1, C syntax can be embedded to access variables from the compiler environment or to add control flow (e.g. conditional computation of a flag). Lines that contain "{" or "}" indicates C code and are not translated by the generator but copied to the output file (itself a C file). Listing 2 shows the output of the generator with Listing 1 as input.

Listing 2: Generator output snippet

```
void op_0x07(int af)
{
        *cur_block.genp++ = PM_MOV(COND_AL,0,0,REG_LSR(5, 7));
        *cur_block.genp++ = PM_ORR(COND_AL,0,5,0,REG_LSL(5, 1));
        if(ALIVE(FC)){
                *cur_block.genp++ = PM_MOV(COND_AL,0,4,REG_LSL(0, 4));
        }
        *cur_block.genp++ = PM_AND_IMM(COND_AL,0,5,5, 0xff);
        cur_block.const0 &= 0x0f;
        cur_block.const1 &= 0x0f;
        cur_block.const0 |= 0xe0;
}
```

To reduce programming effort the generator supports a simple text-based macro mechanism. Macros can be defined with parameters that are handled in a textual search-and-replace way which leaves the contents of a macro open - both C and/or assembler code can be used.

7.5 The Parser

GNU Bison and flex were used to generate a parser which is used to translate a single assembler line as mentioned above. The goal of the implemented grammar was to achieve far-reaching compatibility to commonly used ARM assembler syntax while keeping complexity low. This enabled us to directly reuse parts of the ARM assembler CPU core (using classical emulation) we implemented before.

The representation of immediate values is special: Shifted immediates are expected in binary format by the grammar. Whenever an immediate is expected a C expression may be used instead. The big advantage of this technique is that templates can utilise variables provided by the dynamic compiler as immediates. C macros help to make the usage of shifted immediates in templates more comfortable. Listing 3 is an example that demonstrates various possibilities of embedded C code in templates. In line 1 the macro READB stores an immediate value (of the current target instruction positioned at cur_block.pc) in the globally available C variable "temp". Line 2 evaluates the value of the carry flag: The third line is only relevant if the Carry Flag is not known at (dynamic) compile time. Finally line 7 shows an example of shifted immediate value 0xff00.

Listing 3: Possibilities of Embedded C Code

```
1       {temp=READB(cur_block.pc + 1);}
2       if(!IS_CONSTANT(FC)){
3               sub             r0, r5, 'temp'
4
5       ...
6
7       and             r0, r5, 'IMM_ROTL(0xff, 4)'
```

Another difference to classical ARM assembler syntax is posed by branch instructions. Labelled branch targets (as well as labels in general) are not implemented in our design. As branch targets are relative to the current ARM program counter which may not be known prior to dynamic compile time, absolute branch targets (e.g. a call of a C function) have to be calculated outside the parser.

The employed attributed grammar basically distinguishes between five types of assembler instructions. These are: moves, branches, compare/test instructions, arithmetic and memory operations. The complete attributed grammar can be found in the appendix.

7.6 Compiler

The compiler uses code templates (one for each target instruction) that are stringed together to form a native block. Each code template is a C-function (generated as mentioned above) that receives parameters to produce a specific variant of a target opcode. Technically these functions are organised in a function pointer array that is indexed by the target instructions binary 8-Bit opcode.

In a first step the compiler scans the target instructions (top-down) to find the end of a basic block indicated by a terminating instruction (ENDOP) e.g. jump instruction. Cycles for each target instruction seen are counted. The subsequent pass is bottom-up to enable some liveness optimisations. Finally host instructions are produced in a third top-down pass.

7.6.1 Flag Constant Propagation

Flag Constant Propagation has three aspects:

1. Flags that are unconditionally written to a fixed value by a target instruction (constant flags) do not need to be written within an uninterruptible block but at the end of it.
2. Because constant flags are not written to the virtual flag register, each template has to be parametrised to produce a proper version of the target instructions native equivalent without reading constant flags at runtime. This means that the constant flag vector MUST be considered within the template - the virtual flag register contains no information about these flags (compare with ARM instruction set description [9]).
3. Constant flags must be recalculated by each template at dynamic compile time - this provides the propagation of constant flags.

As constant propagation is mainly located within template functions it is performed in the final top-down pass of the compiler. The algorithm uses two global bit-vectors: const0 and const1. Before any template function is executed the bit-vectors are initialised with zero, meaning no flags are constant. Template functions query this vectors by the C macro IS_CONSTANT. At the end of a template function the values of const0 and const1 need to be recomputed to propagate constant flags to the successive template function. First the bits for flags that are written (either constant or not) are reset in const0 and const1. Second the bits for flags that are written constant are set in the corresponding const bit-vector.

At the end of one basic block where interrupts are possible a consistent virtual state has to be provided by templates that terminate a block (ENDOPs): The compiler function set_constflags is called which simply adds a few instructions that store an accumulation of all prior constant flags at runtime.

7.6.2 Flag Liveness Analysis

As mentioned before flag computations may be unnecessary as flags are overwritten without being read before. Flag liveness analysis has the goal to avoid such computations. It is mainly accomplished in the bottom-up pass of the compiler. An array of flag vectors is filled, which is used in the final (code generating) top-down pass to parametrise the template functions. The basic idea is that each target instruction tells its predecessor which flags are alive and need to be computed. Assume that instructions are numbered ascending inside one block: A flag is alive for an instruction n if it is read by instruction k (with $k > n$) and none of the intermediate instructions i ($k < i < n$) overwrite the flag. Control flow within templates (compare with line 9 of Listing 1) provides that computation of dead flags is not dynamically compiled and therefore not executed at runtime. As the last instruction of one block has no distinct successor (it may be a conditional branch and interrupts are allowed after one block) the compiler has to decide which flags are alive for this instruction. In a first simple approach, all flags are assumed to be alive at the end of the block. A more sophisticated way to find alive flags is to consider all possible successors (this may include interrupt handler functions) and to treat a flag as alive if it is alive in all the possible successors. Applying the rules described above this means that a flag is alive if the first operation on this flag is a read (and dead if it is a write operation) for one successor. As further instructions with more than one successor can appear a tree is spanned. At the moment this lookahead mechanism to find alive flags for the last instruction of a block is not fully implemented.

7.6.3 Block Termination

One of the key issues of the architecture is the handling of basic blocks. They not only need to be dynamically compiled but must be glued together while keeping timing information cycle-accurate. Basic blocks are terminated by so-called ENDOPs. These can be either operations where control flow forks (e.g. conditional jump, computed jump) or such that require block termination for special reasons like interrupts (e.g. halt, enable interrupts).

At the end of one basic block, basically the following tasks must be performed:

1. The size of the basic block, in target CPU clockcycles, must be counted. This is needed for cycle-accurate emulation of the rest of the system.
2. Control has to be passed back to the main loop, which takes care of the former mentioned system emulation.
3. The value of the new target program-counter has to be returned (as it is part of a proper virtual state after one basic block).

We combine these tasks in a structure called linkblock. They may be specific for each ENDOP and are therefore implemented in the template functions. To make programming of these tasks effective, generic support functions located in the compiler are called.

With the forking nature of ENDOPs the need for multiple linkblocks in one native block arises. The decision which one of these linkblocks is actually executed commonly depends on the value of a flag. In other words linkblocks may be skipped which is realised through the insert_skip function that receives the condition as parameter.

During the execution of a native block the host register r9 is reserved for counting remaining target CPU cycles. It is initialised in the control loop to a value that indicates the duration that may be spent in native blocks. A linkblock begins with a call of linkblock_begin (at dynamic compile time) which inserts the host instruction sub to decrease the cycle counting register by the target CPU cycles spent in the current basic block. As ENDOPs may have different cycle counts depending on the chosen linkblock (e.g. conditional branch) the cycles that were counted in the compilers first pass need to be corrected, since by default the highest number of cycles possible for an ENDOP is counted by the compiler. Linkblock_begin gets a parameter of delta cycles to fulfil this correction.

By convention an ENDOP template has to ensure that the value of the next PC resides in temporary register r1 before calling linkblock_end which compiles in the jump back to the control logic.

If an ENDOP template calls insert_skip to conditionally skip a linkblock, first of all a placeholder instruction is inserted in the native block that is subsequently replaced by a conditional branch when calling linkblock_end (the branch offset depends on the previously unknown linkblock size).

7.7 Cascading

Especially for small basic blocks frequent returning to the control logic poses a great overhead. A complete virtual state (including PC) is needed to emulate other system parts and to look up the next basic block. This overhead can even nullify performance gained through dynamic translation. Basic blocks themselves can not be easily enlarged but chained together in order to reduce overhead. The return instruction to the control loop as described before is replaced by a direct branch to the successive native block. This can be done if the successor is known at dynamic compile time (static successor) and pays off through less lookups (PC needs to be returned less often) and system emulation in larger junks.

7.7.1 Backpatching

A technique called backpatching is applied to chain blocks together. Linkblocks with static successors are compiled differently to be prepared for patching: instead of the PC a pointer to a data structure (located at the end of the linkblock) is provided in r1. Besides the PC this data structure contains a pointer (patchpointer) to the patchable area within its linkblock. The return to the control logic is replaced by a branch to a special patching logic. This means that whenever such a linkblock gets executed control is passed to this patching logic (which actually removes the jump to itself). It uses the stored PC to look up the native successors or even compile it to get its start address. Then it inserts a jump instruction to this address at the location stored in the patchpointer.

7.7.2 Dynamic Cascading

Dynamic compilers that operate on user level can handle system calls on a high level. Often target system calls can be mapped to host system calls and host system interrupts suspend the execution of native blocks. For example a timer interrupt can be registered on the host system that interrupts the execution of native blocks after a certain time measured in host system cycles. This may violate relative time coherency in the emulated system. For many applications this is acceptable to a certain degree.

Our target platform has no operating system that abstracts hardware. Therefore high level emulation cannot easily be applied because target applications (especially interrupt handlers) may access hardware in an arbitrary way. Accurate timing is a prerequisite for most target applications found.

Interrupts may occur at every virtual target clock tick and must be handled immediately to guarantee exact emulation. To realise this requirement execution of basic blocks would have to be suspended. This contradicts to the atomic nature of our basic blocks: Within basic blocks the virtual system state is incomplete and would have to be completed on an interrupt.

This is not supported in our design and would require complex changes to our architectural style that seem not too promising to us. Anyway it is worth to be kept in mind for future work.

As mentioned above native blocks are chained together to improve performance - but this virtually increases block size as control is not returned to the control logic between chained basic blocks. To solve the dilemma we decided: The execution of one basic block may not be interrupted. Basic blocks are not cascaded statically but through conditional branches taking into account how many cycles passed since the last invocation of control logic. Together with an upper limit for basic block sizes the resulting loss of accuracy is kept bounded.

The timing accuracy is determined by two parameters: MAX_BASIC_BLOCK_CYC and CASC_CYC. MAX_BASIC_BLOCK_CYC is the upper bound for cycles consumed by one basic block. This means that large basic blocks are broken into several native blocks by the compiler (only the last of these is terminated by an ENDOP).

The control logic initialises the cycle counting register to the value of CASC_CYC before execution of a series of cascaded native blocks is started. Whenever execution reaches the end of a native block the cycle counting register is decreased accordingly. If cycles are remaining (counting register > 0) the conditional branch to the successive block is taken, otherwise the cascade is intermitted. The control logic is able to calculate the target cycles spent in native blocks by subtracting the counting registers value from CASC_CYC.

Maximum cycles spent in a native cascade can be calculated:
CASC_CYC + MAX_BASIC_BLOCK_CYC - 1.

After some instructions that demand immediate interrupt handling the cascade has to be broken in any case. The patching logic inserts an unconditional branch back to the control logic for such instructions (e.g. halt).

Compared with static cascading our dynamic cascading has the disadvantage of producing overhead due to frequent interrupts of the native instruction flow. As a countermeasurement this interceptions are kept as short as possible by eliminating the need for lookups of native successors that point the way back into the chain: The backpatching algorithm inserts a datastructure into the linkblock containing PC and the start address of the native successor. A pointer to this datastructure is passed to the control logic when the native cascade is left.

7.7.3 Loop Detection

An alternative to some cases of backpatching can be realised in the compiler. If the native successor is known it would be possible to directly compile in all instructions needed for cascading at dynamic compile time. This principle could even be applied recursively (a tree of basic blocks would be compiled as a whole). The downsides of such an approach are that "unused" blocks might be compiled, the principle of locality is violated and compile time is not spread over runtime. We found it more elegant to solely make use of backpatching with just one exception: Basic blocks that have themselves as successors are not backpatched but directly modified (within the compiler) to support chaining of "micro loops" even if they are temporary compiled. The start address of such a block is obviously known without a lookup.

7.8 Handling of Interrupts

Fast and accurate interrupt handling is the drive of our architecture. Interrupts have to be handled in a periodic manner between the execution of native blocks which therefore have to provide the target PC as it is expected to be on the stack when the handler routine is entered. It should be noticed at this point that within the control logic either the PC (interrupt occurs) or the native successor (no interrupt occurs) is needed. That justifies the usage of the datastructure in the linkblock described above as it is filled only once during dynamic compiletime and read with just one memory access during runtime. Providing the pointer to this datastructure on return to the control logic does not even consume a single additional cycle: Clever positioning of the datastructure (immediately behind the branch instruction back to control logic) and using a branch with link instruction effects the link register to contain this pointer.

In principal interrupt handlers are compiled as any other code but their a priori known start addresses allow to bypass the lookup mechanism: An array of native addresses (one for each interrupt handler) is kept in memory. On the occurrence of an interrupt a jump to the according native address is performed - in fact a single load to the host systems program counter is executed. Interrupt handlers are not precompiled, instead the array is initialised with addresses of special wrapper routines that facilitate the compiler to translate the corresponding interrupt handler to host code, replace the arrays entry with the resulting start address and branch to it.

7.8.1 Block Size

Speed and accuracy are conflicting goals as increasing block size improves performance while causing lateness and jitter of interrupts and therefore potentially altered program flow. We have empirically determined that 32 cycles is a good choice for both CASC_CYC and MAX_BASIC_BLOCK_CYC keeping nearly every tested target application running correctly. Increasing cycle parameters too much results in dramatic loss of compatibility that often manifests in freezing applications. It should be noticed that 32 cycles is a rather small value that implies highly frequent execution of control logic which justifies greedy assembler code for its implementation.

7.9 Compiler Cache

It is obvious that execution of dynamically compiled code has runtime advantages compared to classical interpreters. Nonetheless dynamic compilation poses an overhead that must be justified with speed gains. In general several executions of compiled code are needed to compensate dynamic compiletime. Therefore compiled blocks have to be kept in a cache. [3] [4]

The caching mechanism must support a mapping of target PC to (host) start addresses of cached native blocks. Two aspects must be taken into consideration: The structure for storage of native blocks and the mechanism to find a specific block, which is in most cases some sort of index. Often only a subset of already compiled blocks can be held in the cache at once. For that reason replacement strategies must be implemented. [1]

7.9.1 ROM Cache

For now we have chosen rather simple caching strategies. In our target system memory can be partitioned - we strictly distinguish between ROM and RAM areas for which we implemented different caching mechanisms according to the nature of these areas. The implementation for the ROM areas is mainly straight-forward. The native blocks are sequentially stored in a fixed size area. As contents of ROM cannot be changed the only reason for invalidating native blocks is a cache overflow (when no more space for newly compiled blocks is available), which is handled through a complete cache flush in our system. ROM is banked in the target system to enlarge address space. For the not switchable bank 0 that spans from the addresses 0x0000 to 0x3fff an ordinary lookup table is used that holds the start address of a native block for each target address. An entry of 0 for a given index (target start address - PC) signalises that no basic block has been compiled for it yet.

All other ROM banks (1-n) share a common target address space reaching from 0x4000 to 0x7fff. A switch of the bank changes the contents in this area which can be seen as a special case of self modifying code as the active bank can only be determined at runtime. This implies that the backpatching algorithm cannot designate a distinct successor at dynamic compile time when cascading into or even within such an area. But this implication can be weakened through the assumption that bank switches do not occur while the program counter resides in a switchable bank. In fact this assumption holds for all tested applications and allows the following implementation:

In the switchable area the backpatching algorithm can unambiguously determine the address of a successive basic block (it chooses the block within the same bank). Cascading from areas where switching is possible (e.g. ROM bank 0) to the switchable area is the only case where no distinct successor is known. We simply do not backpatch in this case - a lookup at runtime is accepted.

The lookup tables for the switchable area are organised two-staged. For each bank a separate lookup table similar to the one described for bank 0 is used. On a lookup in a first step one of these lookup tables is selected according to the current ROM bank number. Afterwards the PC is used to index this table and the start address is resolved.

7.9.2 Ram Cache

RAM areas demand a more complex handling: Before executing a cached native block, it has to be guaranteed that the corresponding target opcodes have not changed since their compilation. [2]

Our target system has several areas of RAM of which only one (the HIRAM) is cached. This HIRAM area is used by nearly all target applications (for technical reasons - when DMA transfers are used) and is only 128 bytes in size. A bitmap is used to mark areas of the HIRAM that are already compiled, one bit represents one HIRAM byte resulting in a bitmap size of 16 byte. Write access to the HIRAM is processed through function calls that check if already compiled bytes are affected - the separate HIRAM cache gets flushed in these cases. Native blocks within the HIRAM cache can be cascaded but due to separation of ROM and RAM caches cascading from or to the HIRAM cache cannot be done easily (independent cache flushes) and is not implemented yet.

Update: The latest version of iBoy supports caching of compiled blocks that belong to all sorts of RAM (it is no longer limited to the small HIRAM area). Each basic block of these areas is compiled and then stored in a separate memory structure (called temp_block), that can be flushed independently. Every such memory location must be big enough to store wort-case sized compiled blocks, which results in some waste of memory. Besides the actual compiled block, this structure holds the target PC of this block, its size and 32-bit checksum.

When a new block in RAM is compiled, the lowest bits of the PC (program counter of target architecture) are used to index an array that consists of temp_blocks. With this simple form of hashing about 256 to 1024 temp_blocks are needed to get an acceptable rate of collisions (different basic blocks share the same temp_block) in practise. After a block is compiled to its temp_block and the PC is inserted, the checksum for this basic block is computed and stored in the temp_block.

Listing 4: checksum computation

```
inline unsigned int compute_crc(int pc, unsigned int len) {
        unsigned int ret=0, temp;

        while(len--) {
                temp = readb(pc++);
                asm ("eor %0,%1,%0, ror #8": "+r" (ret) : "r" (temp));
                // XOR and rotate right 8 Bits
        }
        return ret;
}
```

The checksum is calculated over target opcodes of the basic block. Each byte of this basic block is successively xored on a zero initialised 32-bit value which is rotated right by 8 bits after each iteration (see code).

After the first compilation and initialisation (PC, checksum) the native part of the temp_block can be executed directly. When the same block is to be executed again, the lookup mechanism has to assure that it is still stored in the appropriate temp_block and that the memory containing the corresponding basic block has not changed. This is done by reading the PC stored in the temp_block and computing the checksum of the bytes in RAM occupied by this basic block, which must match the one in the temp_block structure.

To avoid the timeconsuming lookup mechanism as often as possible, the compiler inserts a call to a special function "check_crc" as the very first instruction in all temp_blocks. This function not only checks whether the contents of temp_block are up-to-date but also updates the temp_block if needed calling the compiler. As this function is executed transparently to the temp_block caller (lookup mechanism or predecessor block), it is even possible to link and cascade blocks in RAM area.

8 Summary of Improvements for the Latest Version of iBoy

Besides the two biggest changes involving Gameboy Color support and caching of RAM areas (already described in Section 6.1 and 7.9.2) many other (smaller) changes were made to iBoy that improve performance, compatibility or user interaction.

A new version of the gnu gcc toolchain was used, that solved some problems with memory allocation and improved performance a bit. Besides the 4th generation grayscale iPod, many other iPods are now supported to run iBoy (including those with colour LCDs). An example for improvements in the user interface is a graphical filebrowser.

Most of the hand crafted assembler templates representing target opcodes that read or write to/from memory (especially those that access HIRAM) were optimised. Instead of calling a function that computes the memory location (in upper address space) at runtime (depending on the PC), parts of these computations are done at compiletime.

9 Evaluations

9.1 Correctness

Correctness of software is hard or even impossible to prove, especially for complex software as dynamic translators. But we can give some evidence that indicates some degree of correctness. Two aspects can be taken into consideration. First the introduction of blockbased emulation and second the correctness of the translation of such a block and its execution.

The introduction of blocks as described in section 7.1 changes runtime behaviour. It is desirable that these changes have only marginal or no influence on the behaviour of the full system as percepted by the user. A testing script automatically started the block based emulation system with a wide variety of applications and took a screenshot after a certain amount of time. An inspection of these screenshots and various interactive manual tests lead us to the assumption that with a blocksize of 32 nearly all applications run without noticeable sideeffects.

To enable a stepwise and verifiable implementation of the dynamic binary translator a blockbased version of gnuboy was implemented and used for the first run of compatibility tests described above. To keep emulation running and test the partially implemented binary translator (instructions were implemented one after another) the block based CPU core was used in parallel to the translator. State comparisons ensured that blocks were semantically equivalent processed in both implementations. In case of a discrepancy of states emulation was stopped with debug outputs.

In the final version even excessive testing did not reveal any more discrepancies which leads to the assumption that the implemented dynamic translator is correct, at least to an admissible degree.

9.2 Performance

To measure the performance of the dynamic translator with a high level of objectivity following preparation have been taken: The interface to the graphical subsystem was given an unidirectional data flow to ensure that the main processor (and therefore the CPU emulation) is not slowed down due to backpressure from the LCD routines. A consequence of this is that scanlines are dynamically skipped in case when the graphical subsystem would get a bottleneck.

An automatic mechanism is implemented to measure the realtime passed for a specific period of emulation. This period is defined through a start and a stop frame number. We decided it to be 1001 frames long - that corresponds to approximately 17 600 000 CPU (or Idle) cycles. The measurement does not start at the very beginning of the emulation (first frame) but after 500 frames to avoid influences of initialisation processes in the results.

To have a good reference only the dynamic translating CPU core was replaced by the CPU emulation core of the original gnuboys C implementation - the rest of the system is (except for some minimal needed changes) exactly the same. For all benchmarks the whole sound subsystem was disabled - together with the loosely coupled graphical subsystem this ensures benchmark results mainly correspond to CPU emulation performance.

Figure 5 show the characteristics of ten applications used for our benchmarks. During execution the program counter resides in different memory areas. The diagram shows the ratios of these areas (dynamically counted in cycles within the 1001 benchmarked

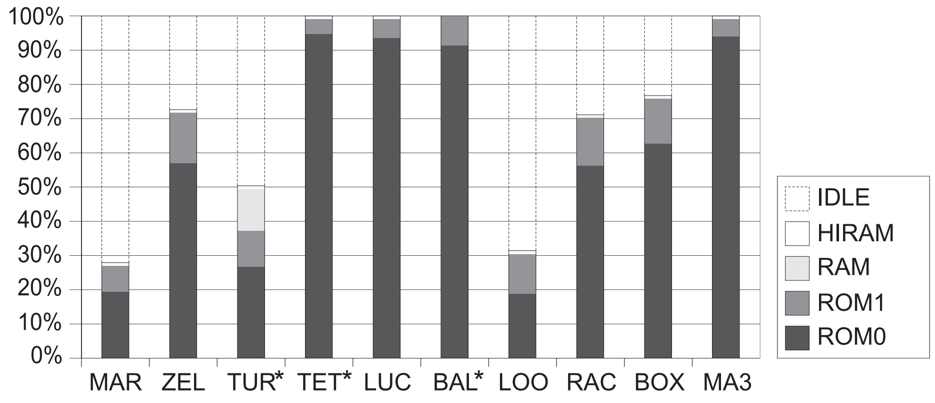

Figure 5: Benchmark applications with used memory areas

frames) which are important to know because different caching strategies (as mentioned earlier) apply for each area. Another important information to be noticed is the idle-ratio of the CPU.

Applications marked with * were measured with user interactions. As this gives slightly different results for each run, the mean of three runs was taken. All other applications have some form of demo mode (typical application scene or intro that produces a representative workload compareable to interactive application usage) where no user interaction is needed.

ROM	MAR	ZEL	TUR	TET	LUC	BAL	LOO	RAC	BOX	MA3
OPS	4.19	3.47	4.82	3.09	3.44	3.87	6.08	4.64	6.53	3.33

Table 1: Blocksizes counted in operations

The sizes for basic blocks were determined dynamically within the 1001 measured frames (of course in a separate run), averaged and shown in table 1.

In figure 6 the performance relative to realtime is shown. Different applications with different optimisation options for dynamic binary translation and the C interpreter are compared with each other. Applications with high idle ratio tend to perform better with all implementations. But simulating idle time is still rather expensive as timers, interrupts and other components have to be emulated making performance gains less than one would expect. With the interpreter realtime is only reached for three of the benchmarked applications; with dynamic translation on the other hand for all of them.

The rate of computed jumps (JP HL and RET instructions in our case) as shown in table 2 has little influence on performance and seems not to allow conclusions to be

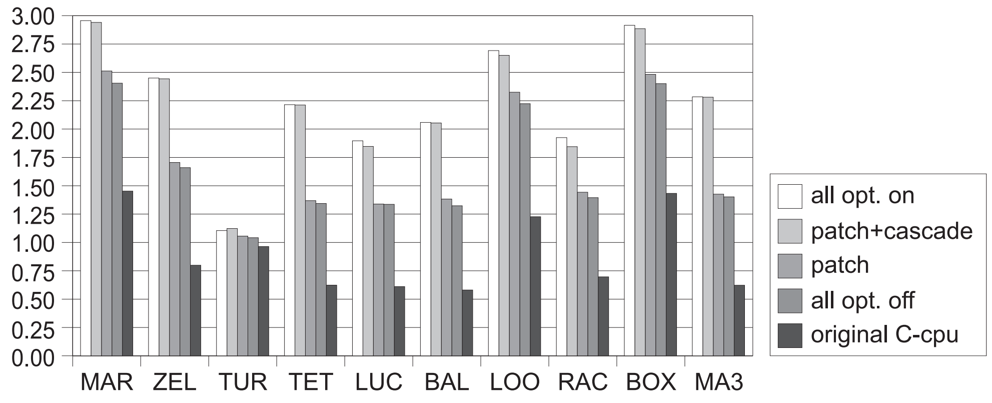

Figure 6: Performance Comparison of Implementations, Realtime=1

ROM	MAR	ZEL	TUR	TET	LUC	BAL	LOO	RAC	BOX	MA3
COMP JP	9.1%	3.8%	17.0%	2.1%	1.3%	3.6%	4.2%	12.7%	15.7%	1.5%

Table 2: Percentage of taken computed jump in relation to all block ends

made directly.

The relative performance (compared to the interpreter) is shown in figure 7. As the emulation of the idle loop is not faster with dynamic translation the most significant performance gains can be reached with applications with no idle times (TET, LUC, BAL, MA3). A special case is the application "TUR" as it is the only one that makes significant usage of RAM for executable code. Yet no caching strategy is implemented at all for the RAM area (as blocks are not cached they need to be compiled every time before they are executed), explaining the poor performance of this application.

Update: This is no longer true for the new version of iBoy where all RAM areas are cached as well (see section 7.9.2). It will be shown later that this significantly influences performance of application "TUR".

Figure 8 shows how optimisation options of dynamic translation increase the performance (in % compared to a version with all optimisations turned off - except constant propagation, which is embedded into opcode definitions and cannot be disabled). The first optimisation that was benchmarked is patching (without cascading). In this case native blocks get patched so that control flow is always returned to the main loop after their execution but the address of the native successor is passed back if possible. Therefore lookups can be omitted for all kinds of uncomputed jumps. Patching in general increases performance only slightly (less than 5%) because lookups are relatively cheap, it might pay off for more complex lookup mechanisms.

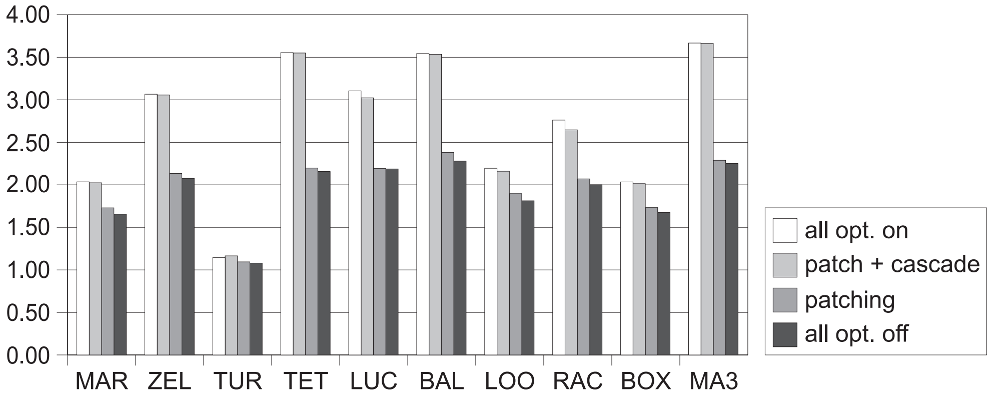

Figure 7: Performance gains through DT, original C Interpreter=1

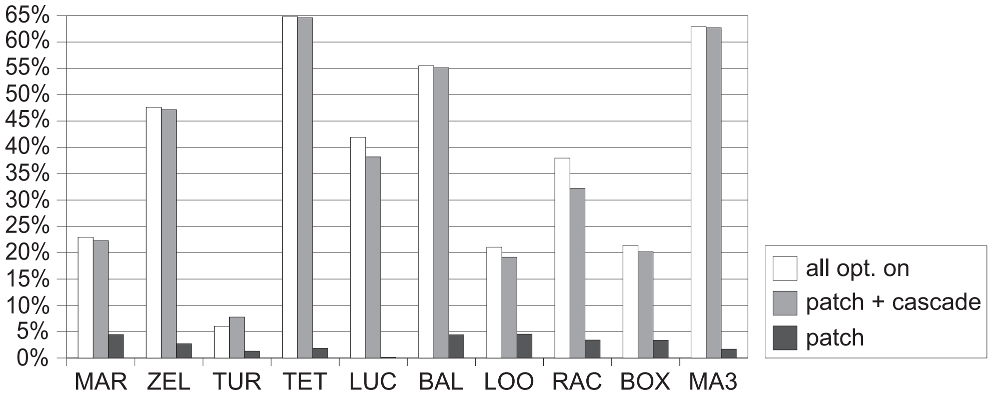

Figure 8: DT speedup through optimisations

The next level of optimisation benchmarked has dynamic cascading turned on (with patching enabled - which is a prerequisite). This virtually increases the blocksize to a given limit (determined by CASC_CYC and MAX_BASIC_BLOCK_CYC). Many returns to the control logic can be omitted, this especially applies for applications with small application specific basic block sizes (e.g. TET). Figure 8 shows the significant speedup due to cascading with CASC_CYC and MAX_BASIC_BLOCK_CYC set to 32. Different characteristic basic block sizes of applications result in different potential for performance gains through dynamic cascading.

The highest level of optimisation (all opt on) has liveness analysis optimisation in addition to those mentioned earlier enabled. Flag liveness speedups are in the order of a few percent (0% to 6% in our case). The performance degree seen at "TUR" is mainly an effect of error of measurements (interactive benchmarking used). The impact of liveness analysis depends on the amount of dead flag computations. As flags are assumed to be alive at the end of blocks larger basic blocks increase the probability for dead flags. Nonetheless application specific code style (that is not measured) has even more influence. Therefore block size does not strictly correlate to effectiveness of flag liveness analysis in our measurements.

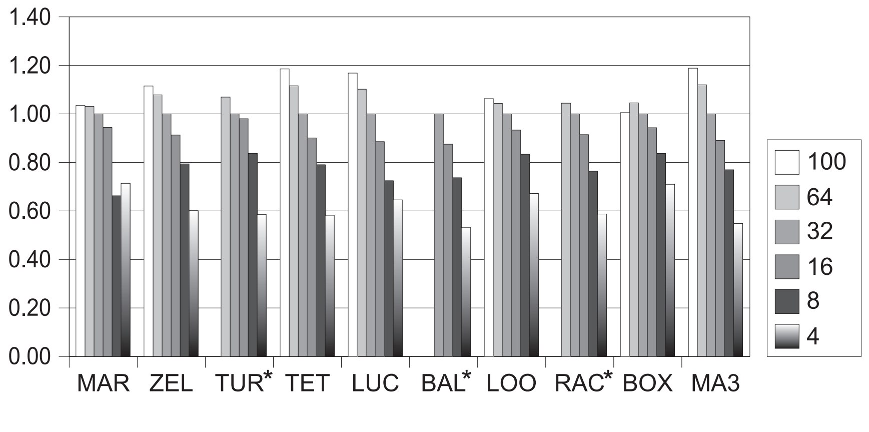

Figure 9: DT with different block sizes, blocksize32=1

Finally figure 9 shows the effect of dynamic block sizes on performance (CASC_CYC = MAX_BASIC_BLOCK_CYC = [4,8,16,32,64,100]). Applications marked with * did not allow meaningful measurements at large dynamic block sizes due to incorrect emulation. In fact the rather small value of 32 cycles for dynamic basic block size seems to be near the upper limit to ensure correct emulation. Increasing dynamic block size seems to increase performance in a logarithmic way. Anyway the actual dynamic block size may vary depending on the amount of computed jumps and other effects (see section 7.7.2). So the results of measurements do not exactly follow this rule.

9.2.1 Performance Impact of RAM Caching

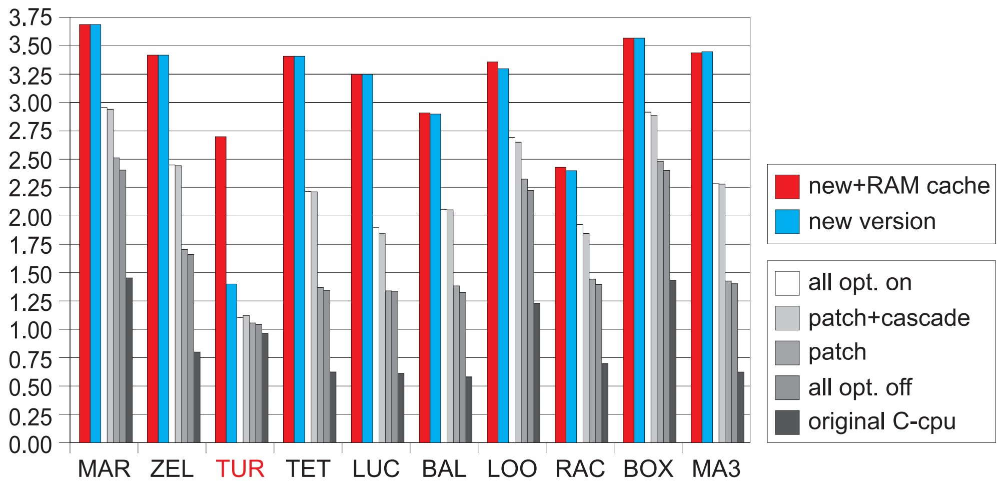

Figure 10: Latest Version of iBoy, RAM Cache

In figure 10 the performance of the general improvements described in 8 and the impact of the RAM caching mechanism can be seen. All values are ratios in respect to realtime (for comparison the performance values of figure 6 are repeated). All benchmarked applications experienced speed improvements in the range from 20% to 80% with this general improvements of the new version of iBoy (a blocksize of 32, and all optimisations) enabled.

The only application that makes significant usage of RAM is "TUR", so this is the only one that benefits from RAM caching. With RAM caching enabled the execution speed of "TUR" is nearly twice as high as without and finally in the range of the other benchmarked applications. The total speedup compared to the old version (white bar) is over 230%. Repeated checksum checks (and possibly collisions in the hashtable) are responsible for the slightly worse performance compared to most of the other applications.

10 Future Work

There are still some parts in iBoy that have to be improved.

The huge lookup tables could be replaced by a more sophisticated hash mechanism resulting in a smaller memory footprint.

A liveness lookahead mechanism could calculate dead flags beyond the scope of one basic block. This could prevent further unnecessary flag calculations. Various optimisations on register level (lifelines, constant propagation)

could further increase performance.

Dynamically changing cascading block sizes might improve performance while keeping the desired level of emulation accuracy. Other more challenging modifications of the dynamic translator could include a trace based approach of caching and optimisation or even a complete new architecture for interrupt handling.

11 Conclusion

Dynamic binary translation can be efficiently applied for system level emulation where applications have direct access to hardware and accurate timing is a prerequisite. It is even applicable for host systems without a memory management unit at least if the emulated target system does not have one either. Compared to a classical target code interpreter our dynamic binary translator executes target code (including interrupts and idle time) at least twice as fast, provided that a caching strategy for RAM areas is implemented. Speedups in this order of magnitude (up to over 350% compared to the interpreter) can even be reached with a lightweighted compiler that sets aside expensive optimisations. A fast assembler optimised interface between dynamically compiled native code and the rest of the emulation system seems to be a key issue for our architecture where execution of native code is tightly intertwined with system emulation.

References

[1] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 1–12, New York, NY, USA, 2000. ACM Press.

[2] Fabrice Bellard. Qemu a fast and portable dynamic translator. In *USENIX 2005 Annual Technical Conference*, pages 41–46. USENIX Association, 2005.

[3] Robert Cmelik and David Keppel. Shade: A fast instruction-set simulator for execution profiling. *ACM SIGMETRICS Performance Evaluation Review*, 22(1):128–137, May 1994.

[4] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 297–302, Salt Lake City, Utah, 1984.

[5] ZiLOG Inc. In *Z80 Family CPU User Manual (UM 008005-0205)*, 2004.

[6] David Keppel, Susan J. Eggers, and Robert R. Henry. A case for runtime code generation. Technical Report UWCSE 91-11-04, University of Washington Department of Computer Science and Engineering, November 1991.

[7] Bich C. Le. An out-of-order execution technique for runtime binary translators. In *ASPLOS-VIII: Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pages 151–158, New York, NY, USA, 1998. ACM Press.

[8] Arm Limited. Instruction cycle timings. In *ARM7TDMI Technical Reference Manual (ARM DDI 0210C)*, pages 139–168, 2001.

[9] Arm Limited. Instruction set summary. In *ARM7TDMI Technical Reference Manual (ARM DDI 0210C)*, pages 30–38, 2001.

[10] David McCullough. Getting started with uclinux. *Cyberguard Corporation–Snapgear Technical Bulletin*, 12, 2001.

[11] Nintendo of America Inc. In *Gameboy (original) Games*, Redmond, Washington, 2004. http://www.nintendo.com/doc/dmg_games.pdf.

[12] Pan of ATX, Marat Fayzullin, Pascal Felber, Paul Robson, and Martin Korth. Pan docs, inner workings on the hand-held game machine known as gameboy, manufactured and designed by nintendo co., ltd. or 'everything you always wanted to know about gameboy, but were afraid to ask'. http://www.work.de/nocash/pandocs.htm, 2001.

[13] Mark Probst. Dynamic binary translation. In *UKUUG Linux Developer's Conference 2002*, 2002.

[14] Mark Probst, Andreas Krall, and Bernhard Scholz. Register liveness analysis for optimizing binary translation. In Elizabeth Burd and Arie van Deursen, editors, *Working Conference on Reverse Engineering (WCRE'02)*, Richmond, October 2002. IEEE.

[15] Mendel Rosenblum, Steven A. Herrod, Emmett Witchel, and Anoop Gupta. Complete computer system simulation: The simos approach. IEEE Parallel and Distributed Technology, Fall 1995.

[16] David Ung and Cristina Cifuentes. Machine-adaptable dynamic binary translation. In *DYNAMO '00: Proceedings of the ACM SIGPLAN workshop on Dynamic and adaptive compilation and optimization*, pages 41–51, New York, NY, USA, 2000. ACM Press.

[17] David Ung and Cristina Cifuentes. Optimising hot paths in a dynamic binary translator. *SIGARCH Comput. Archit. News*, 29(1):55–65, 2001.

[18] Emmett Witchel and Mendel Rosenblum. Embra: Fast and flexible machine simulation. In *Measurement and Modeling of Computer Systems*, pages 68–79, 1996.

Appendix

Listing 5: Lexical Analyzer

```
%option noyywrap

%{
#include <stdio.h>
#include <math.h>
#include "y.tab.h"
%}


W                   [ \t\r\n]+
NL                  [\n]+
COMMENT             "@"[^\n]*"\n"
IMM                 "#0x"([0-9a-fA-F]+)|"#"([-]?[0-9]+)|(\'[^']+\')
REG                 "r"[0]?[0-9]|"r1"[0-5]
ORBA                \[
CBRA                \]
C                   \,

MOV                 "mov"
ARI                 "add"|"sub"|"orr"|"and"|"eor"|"sbc"|"bic"
CMP                 "cmp"|"tst"|"cmn"
H                   "h"
B                   "b"
MEM                 "ldr"|"str"
SHIFT               "lsl"|"lsr"|"asl"|"asr"|"ror"|"rol"
COND                "eq"|"ne"|"cs"|"cc"|"mi"|"pl"|"vs"|"vc"|"hi"|"ls"|"ge"|"lt"|"gt"|"le"|"al"
S                   "s"
BL                  "bl"


%%
{ORBA}   {return OBRA;}
{CBRA}   {return CBRA;}
{IMM}    {
         if(yytext[0]=='\'') {
                 yytext[strlen(yytext)-1]=' ';
         }
         yytext[0] = ' ';
         yylval.str = strdup(yytext);
         return IMM;
         }
{REG}    {
         //yylval.str=strdup(yytext);
         yylval.num=atoi(yytext+1);
         return REG;
         }
{MOV}    {return MOV;}
{ARI}    {yylval.str=strdup(yytext); return ARI;}
{CMP}    {yylval.str=strdup(yytext); return CMP;}

{MEM}    {yylval.num = (yytext[0]=='l'); return MEM;}
{H}      {return H;}
{B}      {return B;}
{SHIFT}  {yylval.str=strdup(yytext); return SHIFT;}
{COND}   {yylval.str=strdup(yytext);return COND;}
{S}      {return S;}
{BL}     {return BL;}
{C}      {return C;}

{COMMENT}           /* remove comment */
{W}                 /* remove ws */

%%
```

Listing 6: Grammar

```
%{
#include <stdio.h>
#include <string.h>

#define YYERROR_VERBOSE

void yyerror(const char *str)
{
        fprintf(stderr,"error: %s\n",str);
}

int yywrap()
```

```
{
        return 1;
}

int main(int argc, char **argv)
{
        if(argc==2)
                stdin = fopen(argv[1], "r");

        yyparse();
}

%}

%token C OBRA CBRA NUM IMM IMMHEX REG SHIFT COND MOV ARI CMP SC BL MEM H S B

%union
{
        int num;
        char *str;
}

%token <num> H S B
%token <num> IMMHEX
%token <str> MOV ARI CMP IMM
%token <str> COND SHIFT CEXP
%token <num> REG
%token <num> BL MEM

%type  <str> asm mem imm
%type  <str> cond flag barrel
%type  <str> mov arit branch cmp
%type  <num> brop memsiz


%%
asms: | asms asm

asm:    mov                     {printf("%s;",$1);}
        |       arit            {printf("%s;",$1);}
        |       cmp             {printf("%s;",$1);}
        |       branch          {printf("%s;",$1);}
        |       mem             {printf("%s;",$1);}

cond:                           { $$ = "COND_AL"; }
        |       COND            { $$ = malloc(100); sprintf($$, "COND_%s", str_toupper($1)); }

flag:                           { $$ = "0"; }
        |       S               { $$ = "1"; }

mov:    MOV cond flag REG C barrel {
            $$ = malloc(100); sprintf($$, "PM_MOV(%s,%s,%d,%s)", $2, $3, $4, $6); }
        | MOV cond flag REG C imm {
            $$ = malloc(100); sprintf($$, "PM_MOV_IMM(%s,%s,%d,%s)",$2, $3, $4, $6); }

arit:   ARI cond flag REG C REG C barrel {
            $$ = malloc(100);
            sprintf($$, "PM_%s(%s,%s,%d,%d,%s)", str_toupper($1),$2, $3, $4, $6, $8); }
        | ARI cond flag REG C REG C imm {
            $$ = malloc(100);
            sprintf($$, "PM_%s_IMM(%s,%s,%d,%d,%s)", str_toupper($1), $2, $3, $4, $6, $8); }

cmp:    CMP cond REG C barrel   {
            $$ = malloc(100);
            sprintf($$, "PM_%s(%s,%d,%s)", str_toupper($1),$2, $3, $5); }
        | CMP cond REG C imm {
            $$ = malloc(100);
            sprintf($$, "PM_%s_IMM(%s,%d,%s)", str_toupper($1),$2, $3, $5); }

mem:    MEM cond memsiz REG C OBRA REG C barrel CBRA {
            $$ = malloc(100);
            switch($3) {
              case 0:case 1: // PM_MEM(cond,L,B,d,n,adr, I,Pre,Upcount(sign),Writeback)
                sprintf($$,"PM_MEM(%s,%d,%d,%d,%d,%s,1,1,1,0)",$2,$1,$3,$4,$7,$9);
                break;
              case 2:case 3:case 4: // PM_MEM2(cond,L,H,S,d,n,adr,I,P,U,W)
                sprintf($$,"PM_MEM2(%s,%d,%d,%d,%d,%d,%s,1,1,1,0)",$2,$1,!($3%2),($3>2),$4,$7,$9);
                break;
            }
        }
        | MEM cond memsiz REG C OBRA REG C imm CBRA {
            $$ = malloc(100);
            switch($3) {
              case 0:case 1: // PM_MEM(cond,L,B,d,n,adr, I,Pre,Upcount(sign),Writeback)
                sprintf($$,"PM_MEM(%s,%d,%d,%d,%d,%s,_0,1,1,0)",$2,$1,$3,$4,$7,$9);
                break;
              case 2:case 3:case 4: // PM_MEM2(cond,L,H,S,d,n,adr,I,P,U,W)
```

ROM	blocksize	snd	livel	patch	pasc	type	time	realtime	comp. to C
MAR							11.62	145%	100%
ZEL							21.12	80%	100%
TUR							17.51	96%	100%
TET							27.09	62%	100%
LUC							27.63	61%	100%
BAL							29.06	58%	100%
LOO							13.76	123%	100%
RAC							24.22	70%	100%
BOX							11.78	143%	100%
MA3							27.10	62%	100%

Table 3: Original C Interpreter

```
                    sprintf($$,"PM_MEM2(%s,%d,%d,%d,%d,%d,%s_0,1,1,0)",$2,$1,!($3%2),($3>2),$4,$7,$9);
                    break;
                }
            }

memsiz:                         { $$ = 0; }
            |       B           { $$ = 1; }
            |       H           { $$ = 2; }
            |       B S         { $$ = 3; }
            |       H S         { $$ = 4; }

barrel: REG { $$ = malloc(100); sprintf($$, "%d", $1); }
            | REG C SHIFT imm {
                    $$ = malloc(100);
                    sprintf($$, "REG_%s(%d,%s)", str_toupper($3),$1,$4); }
            | REG C SHIFT REG {
                    $$ = malloc(100);
                    sprintf($$, "REG_%s_REG(%d,%d)", str_toupper($3),$1,$4); }


branch: brop cond IMM {
                    $$ = malloc(100); sprintf($$, "PM_BRA(%s,%d,%s)", $2, $1, $3); }

imm: IMM    { $$ = check_imm($1); }


brop: B     { $$ = 0; }
        | BL  { $$ = 1; }
%%
```

ROM	blocksize	snd	livel	patch	pasc	type	time	realtime	comp. to C
MAR	100	no	yes	yes	yes	d	5.52	306%	211%
ZEL	100	no	yes	yes	yes	d	6.18	273%	342%
TUR	100	no	yes	yes	yes	i			
TET	100	no	yes	yes	yes	i	6.43	263%	421%
LUC	100	no	yes	yes	yes	d	7.62	222%	363%
BAL	100	no	yes	yes	yes	i			
LOO	100	no	yes	yes	yes	d	5.90	286%	233%
RAC	100	no	yes	yes	yes	d	7.17	235%	338%
BOX	100	no	yes	yes	yes	d	5.76	293%	205%
MA3	100	no	yes	yes	yes	d	6.22	271%	436%
MAR	64	no	yes	yes	yes	d	5.54	305%	210%
ZEL	64	no	yes	yes	yes	d	6.39	264%	331%
TUR	64	no	yes	yes	yes	i	14.29	118%	123%
TET	64	no	yes	yes	yes	i	6.83	247%	397%
LUC	64	no	yes	yes	yes	d	8.08	209%	342%
BAL	64	no	yes	yes	yes	i			
LOO	64	no	yes	yes	yes	d	6.01	281%	229%
RAC	64	no	yes	yes	yes	d	8.40	201%	288%
BOX	64	no	yes	yes	yes	d	5.54	305%	213%
MA3	64	no	yes	yes	yes	d	6.60	256%	411%
MAR	32	no	yes	yes	yes	d	5.71	296%	204%
ZEL	32	no	yes	yes	yes	d	6.89	245%	307%
TUR	32	no	yes	yes	yes	i	15.28	110%	115%
TET	32	no	yes	yes	yes	i	7.62	222%	356%
LUC	32	no	yes	yes	yes	d	8.90	190%	310%
BAL	32	no	yes	yes	yes	i	8.20	206%	354%
LOO	32	no	yes	yes	yes	d	6.27	269%	219%
RAC	32	no	yes	yes	yes	d	8.77	192%	276%
BOX	32	no	yes	yes	yes	d	5.79	292%	203%
MA3	32	no	yes	yes	yes	d	7.39	228%	367%
MAR	16	no	yes	yes	yes	d	6.05	279%	192%
ZEL	16	no	yes	yes	yes	d	7.55	224%	280%
TUR	16	no	yes	yes	yes	i	15.59	108%	112%
TET	16	no	yes	yes	yes	i	8.46	200%	320%
LUC	16	no	yes	yes	yes	d	10.05	168%	275%
BAL	16	no	yes	yes	yes	i	9.37	180%	310%
LOO	16	no	yes	yes	yes	d	6.72	251%	205%
RAC	16	no	yes	yes	yes	d	9.59	176%	253%
BOX	16	no	yes	yes	yes	d	6.14	275%	192%
MA3	16	no	yes	yes	yes	d	8.30	203%	327%
MAR	8	no	yes	yes	yes	d	8.62	196%	135%
ZEL	8	no	yes	yes	yes	d	8.68	194%	243%
TUR	8	no	yes	yes	yes	i	18.25	92%	96%
TET	8	no	yes	yes	yes	i	9.64	175%	281%
LUC	8	no	yes	yes	yes	d	12.28	137%	225%
BAL	8	no	yes	yes	yes	i	11.13	152%	261%
LOO	8	no	yes	yes	yes	d	7.52	224%	183%
RAC	8	no	yes	yes	yes	d	11.48	147%	211%
BOX	8	no	yes	yes	yes	d	6.92	244%	170%
MA3	8	no	yes	yes	yes	d	9.60	176%	282%
MAR	4	no	yes	yes	yes	d	8.00	211%	145%
ZEL	4	no	yes	yes	yes	d	11.46	147%	184%
TUR	4	no	yes	yes	yes	i	26.09	65%	67%
TET	4	no	yes	yes	yes	i	13.09	129%	207%
LUC	4	no	yes	yes	yes	d	13.79	122%	200%
BAL	4	no	yes	yes	yes	i	15.40	110%	189%
LOO	4	no	yes	yes	yes	d	9.33	181%	147%
RAC	4	no	yes	yes	yes	d	14.94	113%	162%
BOX	4	no	yes	yes	yes	d	8.16	207%	144%
MA3	4	no	yes	yes	yes	d	13.49	125%	201%

Table 4: DT with different block sizes

ROM	blocksize	snd	livel	patch	pasc	type	time	realtime	comp. to C
MAR	32	no	no	yes	yes	d	5.74	294%	202%
ZEL	32	no	no	yes	yes	d	6.91	244%	306%
TUR	32	no	no	yes	yes	i	15.03	112%	117%
TET	32	no	no	yes	yes	i	7.63	221%	355%
LUC	32	no	no	yes	yes	d	9.14	185%	302%
BAL	32	no	no	yes	yes	i	8.22	205%	354%
LOO	32	no	no	yes	yes	d	6.37	265%	216%
RAC	32	no	no	yes	yes	d	9.15	184%	265%
BOX	32	no	no	yes	yes	d	5.85	289%	201%
MA3	32	no	no	yes	yes	d	7.40	228%	366%
MAR	32	no	no	yes	no	d	6.72	251%	173%
ZEL	32	no	no	yes	no	d	9.90	171%	213%
TUR	32	no	no	yes	no	i	15.99	106%	110%
TET	32	no	no	yes	no	i	12.33	137%	220%
LUC	32	no	no	yes	no	d	12.61	134%	219%
BAL	32	no	no	yes	no	i	12.21	138%	238%
LOO	32	no	no	yes	no	d	7.26	233%	190%
RAC	32	no	no	yes	no	d	11.70	144%	207%
BOX	32	no	no	yes	no	d	6.80	248%	173%
MA3	32	no	no	yes	no	d	11.84	143%	229%
MAR	32	no	no	no	no	d	7.02	240%	166%
ZEL	32	no	no	no	no	d	10.17	166%	208%
TUR	32	no	no	no	no	i	16.20	104%	108%
TET	32	no	no	no	no	i	12.56	134%	216%
LUC	32	no	no	no	no	d	12.63	134%	219%
BAL	32	no	no	no	no	i	12.75	132%	228%
LOO	32	no	no	no	no	d	7.59	222%	181%
RAC	32	no	no	no	no	d	12.10	140%	200%
BOX	32	no	no	no	no	d	7.03	240%	168%
MA3	32	no	no	no	no	d	12.04	140%	225%
MAR	4	no	yes	yes	yes	d	8.00	211%	145%
ZEL	4	no	yes	yes	yes	d	11.46	147%	184%
TUR	4	no	yes	yes	yes	i	26.09	65%	67%
TET	4	no	yes	yes	yes	i	13.09	129%	207%
LUC	4	no	yes	yes	yes	d	13.79	122%	200%
BAL	4	no	yes	yes	yes	i	15.40	110%	189%
LOO	4	no	yes	yes	yes	d	9.33	181%	147%
RAC	4	no	yes	yes	yes	d	14.94	113%	162%
BOX	4	no	yes	yes	yes	d	8.16	207%	144%
MA3	4	no	yes	yes	yes	d	13.49	125%	201%
MAR	4	no	no	no	no	d	10.31	164%	113%
ZEL	4	no	no	no	no	d	17.36	97%	122%
TUR	4	no	no	no	no	i	28.44	59%	62%
TET	4	no	no	no	no	i	21.50	79%	126%
LUC	4	no	no	no	no	d	21.52	78%	128%
BAL	4	no	no	no	no	i	22.89	74%	127%
LOO	4	no	no	no	no	d	12.13	139%	113%
RAC	4	no	no	no	no	d	19.65	86%	123%
BOX	4	no	no	no	no	d	10.24	165%	115%
MA3	4	no	no	no	no	d	21.59	78%	126%

Table 5: DT with different optimisations

	ROM0	ROM1	RAM	HIRAM	IDLE	SUM
MAR	3406778	1334928	0	170280	12684848	17596834
ZEL	10035326	2580226	0	171140	4821504	17608196
TUR	4682245	1847772	2143943	180704	8718893	17573557
TET	16639427	761957	0	172172	0	17573556
LUC	16439327	962059	0	172172	0	17573558
BAL	16055944	1478566	0	39044	0	17573554
LOO	3302715	2029722	20713	170280	12061356	17584786
RAC	9872447	2467281	0	155380	5080443	17575551
BOX	11040574	2303129	0	166668	4096409	17606780
MA3	16517624	883760	0	172172	0	17573556
SUM	107992407	16649400	2164656	1570012	47463453	175839928

Table 6: CPU Doublecycles spent in different areas