Inter-Task Register-Allocation for Static Operating Systems

Volker Barthelmann
Department of Computer Science II
Universität Erlangen-Nürnberg
Martensstr. 3, 91058 Erlangen, Germany
volker.barthelmann@informatik.uni-erlangen.de

ABSTRACT

In recent years, a growing number of small single-chip embedded systems are used in very high volumes, for example in the automotive industry. Due to the high volumes, these systems are very cost-sensitive. This is one of several reasons why they are more and more using static operating systems. In such systems, all system resources are configured offline and an optimized kernel is generated which is tailored to one specific application.

While this allows the use of operating systems for small ECUs with only very few KB of RAM, it is observed that the memory needed to store task contexts (i.e. register sets) in case of task preemptions makes up a significant part of the RAM needed by the operating system (especially on chips with large register files).

This paper presents ideas and a first implementation on methods to reduce this space through interaction of the compiler and the operating system generation. Together they can calculate an upper bound for the register set that has to be stored for each task. Also, the scope for the register allocator of the compiler can be extended to allocate registers across tasks in order to minimize the total size of RAM needed for task contexts.

Categories and Subject Descriptors

D.3.4 [**Programming languages**]: Processors—*Compilers, Optimization, Run-time environments*; D.4.7 [**Operating Systems**]: Organization and Design—*Real-time systems and embedded systems*

General Terms

Performance

Keywords

register allocation, context-switch optimization, optimizing for space

LCTES'02–SCOPES'02, June 19-21, 2002, Berlin, Germany.

1. BACKGROUND

In recent years there has been a rapidly growing number of small embedded systems used in very high volumes. One example is the automotive industry where the number of Electronic Control Units (ECU) in a single car is approaching 100 for high end automobiles and several dozen in mid-range cars. Another domain is the emerging area of wearable computing which may produce very small systems in high volume.

Typically system-on-chip microcontrollers are used which contain the CPU, peripherals, ROM and RAM on a single chip. ROM sizes of a few hundred KB and RAM sizes of a few KB are common.

Not very long ago, these small systems were typically programmed in assembly language using no operating systems. In recent years, however, high level languages are often being used for such systems and the use of operating systems is getting more common.

As, for example, the ECUs in cars are usually connected via several bus systems and have to co-operate, a common communication layer provided with the operating systems is a big benefit. Also, using the standard features of well-tested operating systems seems to promise more stable systems and shorten development times as application programmers do not have to "re-invent the wheel".

Use of high level languages and operating systems also significantly reduces the amount of work necessary to port an application from one chip or architecture to another. Therefore, chips can be chosen more freely — an important benefit in volume production where every penny counts.

2. STATIC OPERATING SYSTEMS

Classical operating systems are not well suited for such systems due to several reasons. The most important reason is the high memory requirement. Another problem is the danger of fragmentation or memory leaks. Many ECUs are safety-critical and the risk of failure to allocate a system resource due to lack of RAM or memory fragmentation cannot be tolerated.

Static operating systems help to avoid these problems. In such operating systems all system resources like tasks, messages, mutexes etc. used by the application are configured statically using an offline tool. After verifying validity, a kernel is generated which is specially optimized to this very application.

All resources needed by the operating system are statically allocated during generation time and therefore no dynamic memory management is needed. By using this mechanism it

is ensured that allocating an operating system resource can never fail!

As all resources used by the application are known when the kernel is generated, it is possible to produce a very small optimized kernel. Functionality not needed by this application can automatically be omitted, and the size of data structures can be reduced. For example, the data type for a task ID can be a single byte index into different arrays rather than a pointer to a task control block as used in many systems.

This can make a big difference for very small systems and also helps putting all constant data into ROM rather than into RAM.

These benefits of static operating systems allow them to be used in very small systems (e.g. 8bit microcontrollers with 32KB ROM and 1KB RAM) and still leave enough room for the application.

The market of static operating systems is currently dominated by systems conforming to the OSEK specification [16]. This is an open standard (which is currently in the process of ISO standardisation) created by the automotive industry. OSEK is not a certain implementation but it specifies the behaviour and services of operating systems. Several vendors offer OSEK compliant operating systems, and ECUs running such OSEK implementations are used in production cars. A standard file format (OIL – OSEK Implementation Language) is used to specify tasks, scheduling properties and all other operating system resources used by the application.

3. TASK CONTEXTS

While it is possible to use only very few bytes of RAM for most operating system resources, it turns out that the memory needed to store task contexts often makes up the biggest part of RAM usage by the operating system.

When a task is preempted and the operating system switches to another task, it has to store all information needed to switch back to the preempted task later on. Classically this means to store the entire register set of the processor which is available to applications. Interrupt service routines can be viewed similarly as tasks in this respect.

The following microcontrollers used in volume production in cars illustrate that the task contexts can use a significant amount of available RAM (consider that 20 – 30 tasks and interrupt service routines are common figures):

- Motorola MPC555 [15]
  26KB RAM
  32 general-purpose-registers (32bit)
  32 floating-point-registers (64bit)
  ⇒ task context ca. 384 Bytes
- Infineon C164CI [14]
  2KB RAM
  16 general-purpose-registers (16bit)
  ⇒ task context ca. 32 Bytes

As mentioned above, these chips are actually used in high-volume production with static operating systems. While there are still many small microcontrollers using an accumulator architecture offering only two or three registers, even 8bit architectures are starting to use large register sets in order to better suit compiler-generated code:

- Atmel AVR [12]
  128 Bytes – 4KB RAM
  32 general-purpose-register (8bit)
  ⇒ task context ca. 32 Bytes

It can be observed that new architectures (even smallest ones) tend to have many registers. This trend will probably hold on (see [6]) and in some areas processors with even larger register sets are used (e.g. 128 registers, see [17]). When storing a large register set, it is not unlikely to save registers which do not (and maybe can not) contain a live value or are not modified by the preempting tasks.

Many embedded systems have a relatively large number of event-triggered real-time tasks or interrupt service routines which execute only very small pieces of code (e.g. fetching a value from a bus). Compiler optimizations which use a lot of registers (inlining, unrolling, software-pipelining) are rarely used as they often increase code size. Therefore, a large part of the tasks may use only a small part of the register set.

As a result, many systems actually waste valuable RAM for task contexts which could be optimized. Imagine a CPU with 1000 registers: Using conventional techniques, it would need a lot of extra RAM for task-contexts although it might in fact never be necessary to store any registers at all during to a context-switch.

4. CURRENT PRACTICE

As the size of task-contexts actually matters in practice (a few bytes of RAM may cost millions of dollars in high volume production), there are already a few attempts to address this issue.

There have been several approaches to improve context-switch times either by software or hardware. However, they are tailored to much bigger systems and do concentrate on speed rather than RAM usage (see e.g. [9], [1], [8]), or they require a very specific class of applications (see e.g. [4]).

Some systems allow to specify reduced register sets for some tasks. A common variant is an attribute to specify that a task does not use floating-point registers. However, this solution is neither very fine-grained nor very safe without compiler support (consider compilers which use floating-point registers for block-copy).

5. CONTEXT OPTIMIZATION

The idea proposed in this paper is to improve this situation by interaction between the compiler and the static operating system. As the system is static and all tasks are known at generation time, it is possible to save different register sets for different tasks.

When a task is preempted, only registers which contain live values and can be destroyed (by preempting tasks) have to be saved. With knowledge on the operating system, the compiler can determine safe bounds of register sets which have to be stored for each task. Furthermore, changes of register-allocation may result in smaller task-contexts without sacrificing intra-task code quality.

Propagating this information back to the operating system generator allows to allocate only as much RAM as needed by the minimized task-contexts. Obviously, the operating system code which performs context-switches will have to be generated accordingly to save/restore different register-sets depending on the preempted task.

For this purposes an embedded system using a static operating system can be modelled using the set of tasks

$T = \{t_1, ..., t_n\}$, the register set $R = \{r_1, ..., r_m\}$ and a set of code blocks $L = \{l_1, ..., l_k\}$ (see below for details).

5.1 Example

Consider the following (admittedly very small) example of a system with three tasks. Assume fixed priority fully preemptive scheduling with the task priorities given in the code. This implies that task `t1` can be interrupted by tasks `t2` and `t3`, task `t2` can be preempted only by task `t3` and task `t3` is the highest priority task which can never be interrupted.

To illustrate some of the situations that can arise, task `t2` contains a critical section (for example obtaining a mutex using some kind of priority ceiling or priority inheritance protocol) which prevents it from being preempted by task `t3` inside this critical section. Also, tasks `t2` and `t3` share some code, namely the function `f`.

Other situations which could lead to different combinations of preemptions would be tasks entering a blocked state (for example, waiting for a semaphore). All these situations can be formalized using an interference graph which will be described below.

The `alloc` and `free` comments shall indicate the beginning and end of register live ranges. `l1` – `l5` are placeholders for blocks of code which do not change preemptability. These blocks do not have to be straight-line basic blocks but can be rather arbitrary pieces of code as long as the preemptability does not change inside. Of course, a conservative estimate could be used for an entire task, but this could negatively affect the benefits of the optimization. Apparently, this partitioning into code blocks depends on the scheduling mechanism and system services provided by the operating system.

`r1` – `r8` designate the register set. For example, `r7` and `r8` could be floating-point registers (to explain why they are used rather than `r2` and `r3`)in task `t1`.

```
TASK(t1) /* prio=1 */
{
  /* alloc r1, r7, r8 */
  l1
  /* free r1, r7, r8 */
}
TASK(t2) /* prio=2 */
{
  /* alloc r1 */
  l2
  EnterCriticalSection();
    /* alloc r2, r3 */
    l3
    /* free r2, r3 */
  LeaveCriticalSection();
  f();
  /* free r1 */
}
TASK(t3) /* prio=3 */
{
  /* alloc r1, r2, r3 */
  l4
  f();
  /* free r1, r2, r3 */
}
void f()
{
  /* alloc r4 */
  l5
  /* free r4 */
}
```

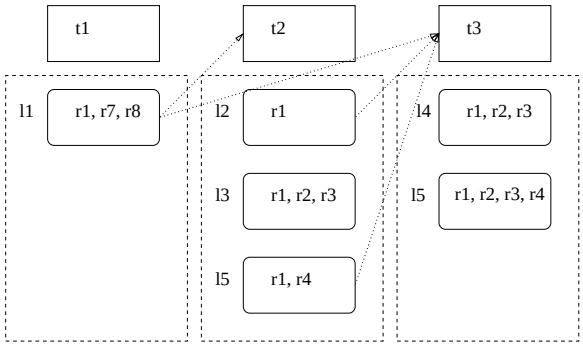

Figure 1: Interference graph

The situation is illustrated in the figure. There are three columns, one for each task. In each column there are all the code blocks `l1` – `l5` which are executed in this task, together with all registers that are live within each block. Note that the live registers are task-specific for shared code. There are different registers live in `l5` (i.e. the function `f`) because a different set of registers is live at the different call sites of `f`.

As a result, the set of used (or live) registers at each block (as it will be calculated by the compiler) is a mapping from a pair consisting of a task and a code block to a set of registers:

$$U : T \times L \to \mathcal{P}(R),$$
$$(t, l) \mapsto \{r \in R : r \text{ is live in block } l \text{ in task } t\}$$

Additionally, there are edges from every code block in a column to all the tasks which can preempt the task within this block. For example, task `t2` can be preempted in blocks `l2` and `l5` by task `t3` but not in block `l3` due to the critical section.

This interference graph is a mapping from a pair consisting of a task and a code block to a set of tasks:

$$I : T \times L \to \mathcal{P}(T),$$
$$(t, l) \mapsto \{t' \in T : t' \text{ can preempt } t \text{ in block } l\}$$

Different scheduling algorithms and operating systems can be modelled that way and will have significant impact on the interference graph.

5.2 Bounding task-contexts

With the model described above, it is possible to specify and calculate an optimized context for each task. First, the set $D(t)$ of the registers each task destroys is needed and can be calculated as

$$D(t) := \bigcup_{l \in L} U(t, l).$$

For the small example, one obtains:
$D(t_1) = \{r1, r7, r8\}$,
$D(t_2) = \{r1, r2, r3, r4\}$,
$D(t_3) = \{r1, r2, r3, r4\}$

When a task t is preempted, it is necessary to store all those registers which are live *and* can be destroyed by any task which can preempt task t. It would be possible to store different register sets depending on the code block where the task was preempted (by looking at the program counter when doing a context-switch). However, this is unlikely to give much benefit and will significantly complicate the context-switch in the operating system.

Therefore, for each task it is sufficient to traverse all blocks of its code, and add to its context all registers which are live in that block and can be destroyed by any task that can preempt it (i.e. there is a corresponding edge in the interference graph). Formally, the task-context $C(t)$ of a task t (i.e. the set of registers that is sufficient to save whenever task t is preempted) can be written as:

$$C(t) = \{r \in R : \exists l \in L, t' \in I(t,l) : r \in U(t,l) \cap D(t')\}.$$

For the small example, one obtains:
$C(t_1) = \{r1\}$,
$C(t_2) = \{r1, r4\}$,
$C(t_3) = \emptyset$

Only memory to store three registers is needed. Without this analysis every task-context would need to provide space for the entire register set ($3 \cdot 8$ registers in this example). Obviously the benefit will often be much smaller, but in very cost-sensitive and already optimized systems, a few bytes saved might actually help to fit an application into a smaller chip and save a lot of costs.

5.3 Inter-task register-allocation

So far, the space for task-contexts has been minimized by analysing the code already produced for the application. The next goal is to further minimize the RAM requirements by considering the task-contexts already when generating the application code, especially when assigning registers.

The scope of register-allocation in compilers varies from single expressions or basic blocks to single functions or inter-procedural register-allocation (see e.g. [3], [10], [5], [7]). In this paper, the scope shall be extended to inter-task register-allocation. Similar to inter-procedural assignment of registers which helps to reduce spilling of registers across function-calls, inter-task assignment can help to reduce the memory needed to store registers of preempted tasks.

The goal of this optimization is to minimize the total space of all task-contexts of a system. As the task-contexts of tasks which cannot preempt each other (for example tasks with the same priority that cannot be blocked) can use the same memory, the space required to store all task-contexts is not necessarily the sum of the sizes of all contexts.

Let $s(r), r \in R$ be the memory requirement of each register, and $\{T_1, ..., T_n\}$ a partitioning of T, such that all tasks in a partition T_i cannot preempt each other, i.e.:

$$\forall t \in T_i, l \in L : I(t,l) \cap (T_i \setminus t) = \emptyset.$$

Therefore, if $M(i)$ is the size needed to store the largest task-context in a partition T_i, i.e.

$$M(i) := \max_{t \in T_i} \sum_{r \in C(t)} s(r),$$

then the object of minimization is:

$$\sum_{i=1}^{n} M(i).$$

Inter-task register-allocation should not negatively affect the intra-task code-generation. Typically, it will only guide the choice between otherwise identical registers.

For the small example presented above, a possible improvement would be to replace r_1 by r_5 in t_2 and by r_6 in t_3 (assuming these registers are available). This would minimize the task-contexts to:
$C(t_1) = \emptyset$,
$C(t_2) = \{r_4\}$,
$C(t_3) = \emptyset$
Although more registers are used, the total RAM requirements would be reduced.

Unfortunately, this optimization problem is not easily solvable (as it is known, even optimal intra-task register-allocation is usually NP-complete as it is at least as hard as graph-coloring). Therefore, it is necessary to find approximations or solutions for special cases. The scheduling algorithm and system services offered by the operating system may affect inter-task register-allocation in a non-trivial way. A first experimental implementation for one specific scheduling strategy will be described below.

6. REQUIREMENTS ON COMPILERS

To carry out the optimizations described in this paper, a compiler has to be able to calculate good bounds on the registers used in different blocks of a task. This can only be achieved if a call-tree can be constructed and the registers used are known to the compiler most of the time. Where this is not possible, worst-case assumptions have to be made and good results are hard to obtain.

Applications using static operating systems usually are rather well suited to this kind of static analysis. Neither recursions nor dynamic memory allocations are usually used due to reasons of safety and efficiency. Also, function pointer variables are generally not used and use of external library functions is very limited (source code is generally available for the entire system).

These restrictions reduce some of the most difficult problems for static analysis. However, there are still a number of requirements on compilers to obtain good results:

- Cross-module analysis is needed as the applications are usually split across files.
- A call-tree has to be built, usually requiring data-flow- and alias-analysis.
- Tasks, the scheduling-mechanism and the operating system services have to be known to enable construction of the interference graph.
- Side-effects (especially register-usage) of inline-assembly (if available), library- and system-functions should be known.

While a few of these features are not yet common in most compilers, more and more modern compilers provide at least the infrastructure (for example, cross-module-optimizations) to incorporate them.

7. IMPLEMENTATION OF CONTEXT-SWITCHES

To make use of the information collected by the compiler, the generation of the operating system has to be adapted. Rather than allocating full contexts for every task, memory for the minimized contexts has to be allocated.

The routines for saving/restoring task-contexts have to be modified in the operating system. Rather than using the same routines for every task, different routines may have to be generated for every task. This will increase code-size to reduce RAM requirements. As RAM is typically much

more expensive than ROM (10-times might be a reasonable estimate), it is still often worthwile to trade in RAM for ROM.

For a set of tasks with similar contexts, it is possible to use the union of these contexts for all tasks in the set and share the routines for saving/restoring the context. This enables fine-tuning between RAM and ROM requirements. As RAM and ROM sizes of a certain microcontroller are fixed, the ability to perform this tuning can be very useful when fitting an application into a certain chip.

If task-contexts are subsets of another one, it may be possible to use the same routines, just with different entry-points. Also, some architectures (e.g. ARM [11] or 68k [15]) have instructions which can save/restore arbitrary register sets controlled by a bit-mask in the instruction code. In such cases, the ROM overhead can be very small.

8. FIRST IMPLEMENTATION AND RESULTS

A first experimental implementation of inter-task register-allocation and minimization of task-contexts has been implemented in an existing C compiler ([13]) which offers the required features mentioned above. Among several back-ends was the MPC555 (PowerPC architecture) which was used for first tests.

The operating system model supported is a fixed priority fully preemptive scheduler without blocking or dynamic priority changes. The tasks are marked with special attributes specifying their priority. These attributes can be created from the application's configuration data (for example OIL, as mentioned above). They provide the compiler with all information necessary to perform the analysis and optimizations mentioned in this paper (e.g. construction of the task interference graph).

The normal intra-task (but inter-procedural) register-allocation was extended to use a priority for each register. If a choice between several otherwise identical registers has to be made by the intra-task register-allocator, it will use the register with the highest priority. Additionally, the top-level function of a task will never save any (callee-save) registers like a normal function.

The inter-task register-allocation modifies these register priorities for the intra-task allocator. It processes the tasks in priority order and adjusts the priorities in such a way that tasks on the same priority (which can share the same context) prefer to use the same registers, whereas tasks on different priorites tend to use different registers.

While the scheduling model considered here is rather simple, it seems possible to extend this mechanism to more complicated schedulers without too much additional effort. Also, many systems are actually using such schedulers, especially if they have to meet hard real-time contraints ([2]).

To obtain some first benchmarks, different combinations of tasks out of the following categories have been created and optimized.

rbuf: A simple task which just fetches a value from an IO port and stores it into a ring-buffer. It uses three general-purpose-registers.

mm: Normal floating-point matrix-multiplication. It uses eleven general-purpose-registers and three floating-point-registers.

n_{rbuf}	n_{mm}	n_{int}	n_{all}	RAM_{std}	RAM_{opt}	savings
10	0	0	0	1040	16	98%
0	10	0	0	3360	296	91%
0	0	10	0	1040	936	10%
0	0	0	10	3360	3024	10%
2	2	2	4	2432	1816	25%
4	2	2	2	1968	1168	41%
4	4	2	0	1968	312	84%
6	0	4	0	1040	384	63%
0	6	0	4	3360	1344	60%
3	1	6	0	1272	596	53%

Table 1: Benchmark results

int: A task using all general-purpose-registers.

all: A task using all general-purpose-registers as well as all floating-point-registers.

Classical optimizations like common-subexpression-elimination, loop-invariant code-motion or strength-reduction have been performed. Loop-unrolling has been turned off. The following table lists the total context sizes with and without optimization. The first four columns (n_{rbuf}, n_{mm}, n_{int} and n_{all}) show how many tasks of each category are used for a test case. All tasks have different priorities. The RAM requirements of each task (e.g. stack-space) are not affected by the context-optimization.

The fifth column (RAM_{std}) lists the total task-context size in bytes with conventional allocation. It is assumed that tasks which do not use floating-point are marked accordingly by the application. A smaller context is allocated for these tasks, even without optimization.

Only the general-purpose- and floating-point-registers which are available for the application have been considered. Any special-purpose registers or registers that must not be used by the application are ignored here. As a result, a full context of a task not using floating-point needs 104 bytes and a full context of a task using floating-point needs 336 bytes. Therefore, if n_f denotes the number of tasks using floating-point and n_i the number of tasks using only general-purpose registers, the non-optimized context size can be calculated as

$$n_f \cdot 336 + n_i \cdot 104.$$

The sixth column (RAM_{opt}) lists the total task-context size using the minimized register sets obtained from the compiler. Inter-task register-allocation was performed, but with this simple scheduling model, it gives additional benefit only in very rare cases. Finally, the last column lists the savings in percent.

It can be observed that the savings depend a lot on the constellation of tasks. As long as almost every task uses all registers, the benefit will be small. However, with every task that uses only a part of the register set, memory is saved.

The first four rows are rather academical as they use 10 tasks of the same category. However, the remaining rows could perhaps reflect typical systems with a number of tasks using the entire register sets as well as some smaller light-weight tasks using only part of the register set.

For several of these constellations, the optimization reduces the RAM usage for task-contexts significantly. Tests with real applications should be performed to verify these results for practical use.

9. CONCLUSION AND FUTURE DIRECTIONS

The results of these first studies have shown potential for this kind of optimization. In a very cost-sensitive and already highly optimized environment, inter-task register-allocation and context-optimization of static operating systems can sometimes further reduce RAM requirements significantly.

The optimization, however, is not easy to achieve. An advanced compiler framework as well as sophisticated operating system technology is needed and must interact. These technologies do exist and are already in use, but the interaction is not yet common. However, embedded operating systems (especially those that are delivered in source code) are already often tied to specific compilers. The vendors of compilers and operating systems tend to cooperate somewhat. Therefore, deeper interactions like proposed in this paper do not seem to be out of reach.

To get further figures on the effectiveness, it is planned to implement these optimizations in commercial static operating systems and, if possible, study the results on real production code. Implementations for more complicated scheduling algorithms and operating systems should be implemented. Tests for different CPU architectures might also yield interesting insight into the effectiveness of the proposed optimizations.

10. REFERENCES

[1] T. Baker, J. Snyder, D. Whalley. Fast Context switches: Compiler and architectural support for preemptive scheduling, *Microprocessors and Microsystems*, 1995, pp. 35–42.

[2] L. P. Briand, D. M. Roy. Meeting Deadlines in Hard Real-Time Systems, *IEEE Computer Society Press*, 1999.

[3] P. Briggs. Register Allocation via Graph Coloring, *thesis*, Houston, 1992.

[4] A. Dean, J. P. Shen. Hardware to Software Migration with Real-Time Thread Integration, *EuroMicro Workshop on Digital System Design*, 1998.

[5] C. Fischer, S. Kurlander. Minimum Cost Interprocedural Register Allocation, *Symposium on Principles of Programming Languages*, 1996, pp. 230–241.

[6] D. Greene, T. Mudge, M. Postiff. The Need for Large Register Files in Integer Codes, 2000.

[7] S. Muchnick. Advanced compiler design and implementation, *Morgan Kaufmann Publishers*, 1997.

[8] P. Nuth. The Named-State Register File, *thesis*, MIT, 1993.

[9] C. Waldspurger, W. Weihl. Register Relocation: Flexible Contexts for Multithreading, *Proceedings of the 20th Annual International Symposium on Computer Architecture*, May 1993.

[10] D. Wall. Global Register Allocation at Link Time. *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pp. 264–275.

[11] `http://www.arm.com`

[12] `http://www.atmel.com`

[13] `http://www.compilers.de/vbcc`

[14] `http://www.infineon.com`

[15] `http://www.motorola.com`

[16] `http://www.osek-vdx.org`

[17] `http://www.trimedia.com`