Chapter 7
Optimization

Embedded systems have to be efficient (at least) with respect to the objectives considered in this book. In particular, this applies to resource-constrained mobile systems, including sensor networks embedded in the Internet of Things. In order to achieve this goal, many optimizations have been developed. Only a small subset of those can be mentioned in this book. In this chapter, we will present a selected set of such optimizations. This chapter is structured as follows: first of all, we will present some high-level optimization techniques, which could precede compilation of source code or could be integrated into it. We will then describe concurrency management for tasks. Section 7.3 comprises advanced compilation techniques. The final Sect. 7.4 introduces power and thermal management techniques.

As indicated in our design flow, these optimizations complement the tools mapping applications to the final systems, as described in Chap. 6 and as shown in Fig. 7.1. Mapping tools may be optimizing, and optimization techniques may involve scheduling. Hence, the scopes of the current and of Chap. 6 are partially overlapping. The focus of Chap. 6 is on fundamental knowledge for mapping to platforms, while the current chapter deals mostly with improvements over basic techniques and is similar to the character of an elective.

7.1 High-Level Optimizations

In the next section, we will be considering optimizations which can be applied to the source code of embedded software, before compilation or during early compilation phases. Detecting regular structures such as array access patterns may be easier at the source code level than at the machine code level. Also, optimization effects can usually be expressed by rewriting the source program, i.e., the modified code can be expressed in the source language. This helps in understanding the effect of such transformations. We do also consider cases in which it may be necessary to annotate

© The Author(s) 2021

P. Marwedel, *Embedded System Design*, Embedded Systems,
https://doi.org/10.1007/978-3-030-60910-8_7

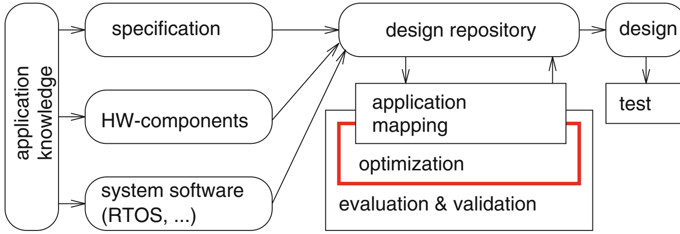

Fig. 7.1 Context of the current chapter

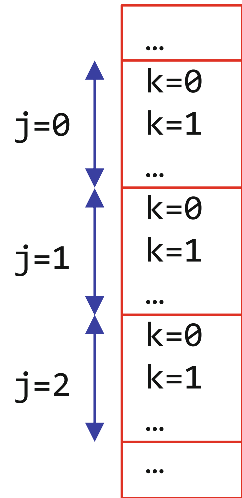

Fig. 7.2 Memory layout for two-dimensional array `p[j][k]` in C

the source code with compiler directives and hints. Such code transformations are called **high-level optimizations**. They have the potential to improve the efficiency of embedded software.

7.1.1 Simple Loop Transformations

There are a number of loop transformations that can be applied to source code. The following is a list of standard loop transformations:

- **Loop permutation:** Consider a two-dimensional array. According to the C standard [289], two-dimensional arrays are laid out in memory as shown in Fig. 7.2. Adjacent index values of the second index are mapped to a contiguous block of locations in memory. This layout is called **row-major order** [405].
  For row-major layout, it is usually beneficial to organize loops such that the last index corresponds to the innermost loop. Note that the layout for arrays is different for Fortran: adjacent values of the first index are mapped to a

contiguous block of locations in memory (**column-major order**). Switching between publications describing optimizations for Fortran and for C can therefore be confusing.

Example 7.1 The following is a loop permutation:

```
for (k=0; k<m; k++)                for (j=0; j<n; j++)
  for (j=0; j<n; j++)       ⇔        for (k=0; k<m; k++)
    p[j][k]= . . .                     p[j][k]= . . .
```

Such permutations may have a positive effect on the reuse of array elements in the cache, since the next iteration of the innermost loop will access an adjacent location in memory. ∇

Caches are normally organized such that adjacent locations can be accessed significantly faster than locations that are further away from the previously accessed location. In this way, caches are exploiting **spatial locality**.

Definition 7.1 Consider memory references to memory addresses a and b. Suppose that we assume an access to a. We observe **spatial locality** if—under this condition—the probability of also accessing b increases for small differences of addresses a and b.

- **Loop unrolling:** Loop unrolling is a standard transformation creating several instances of the loop body.

  Example 7.2 In this example, we unroll the loop:

```
for (j=0; j<n; j++)               for (j=0; j<n; j+=2)
  p[j]= . . . ;            ⇔        {p[j]= . . . ;
                                      p[j+1]= . . . }
```

  In this particular case, the loop is unrolled once. ∇

  The number of copies of the loop is called the **unrolling factor**. Unrolling factors larger than two are possible. Unrolling reduces the loop overhead (less branches per execution of the original loop body) and therefore typically improves the speed. As an extreme case, loops can be completely unrolled, removing control overhead and branches altogether. Unrolling typically enables a number of following transformations and may therefore be beneficial even in cases where just unrolling the program does not give any advantages. However, unrolling increases code size. Unrolling is normally restricted to loops with a constant number of iterations.
- **Loop fusion, loop fission:** There may be cases in which two separate loops can be merged, and there may be cases in which a single loop is split into two.

Example 7.3 Consider the two versions of the following code:

```
for (j=0; j<n; j++)                    for (j=0; j<n; j++)
 p[j]= . . . ;                          {p[j]= . . . ;
for (j=0; j<n; j++)          ⇔           p[j]=p[j]+ . . . }
 p[j]=p[j]+ . . .
```

The left version may be advantageous if the target processor provides a zero-overhead loop instruction which can only be used for small loops. Also, the left version may provide good candidates for unrolling, due to the simple loops. The right version might lead to an improved cache behavior (due to the improved locality of references to array p) and also increases the potential for parallel computations within the loop body. As with many other transformations, it is difficult to know which of the transformations leads to the best code. ∇

7.1.2 Loop Tiling/Blocking

Since small memories are faster than large memories (see p. 170), the use of memory hierarchies may be beneficial. Possible "small" memories include caches and scratchpad memories. A significant reuse factor for the information in those memories is required. Otherwise the memory hierarchy cannot be exploited.

Example 7.4 Reuse effects can be demonstrated by an analysis of the following example. Let us consider matrix multiplication for arrays of size N × N:

```
for (i=0; i<N; i++)
  for(j=0; j<N; j++)    {
      r=0;
      for (k=0; j<N; k++)
         r+=X[i][k]*Y[k][j];
      Z[i][j]=r;
  }
```

Scalar variable r represents Z[i,j] in all iterations of the innermost loop. This is supposed to help the compiler to allocate this element temporarily to a register.

Let us consider access patterns for this code, as shown in Fig. 7.3. We assume that array elements are allocated in row-major order (as it is standard for C).

This means that array elements with adjacent row (right most) index values are stored in adjacent memory locations. Accordingly, adjacent locations of X are fetched during the iterations of the innermost loop. This property is beneficial if the memory system uses prefetching (whenever a word is loaded into the cache, loading of the next word is started as well). Accesses to Y do **not** exhibit spatial locality. If the cache is not large enough to hold a full cache row, every access to Y will be a cache miss. Hence, there will be N^3 references to elements of Y in main memory.

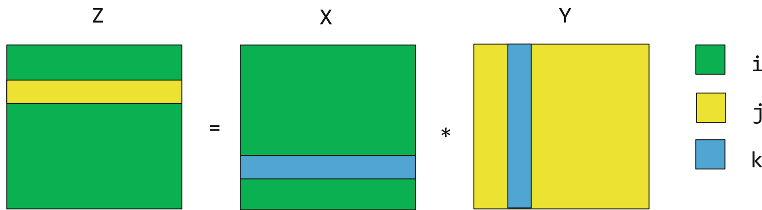

Fig. 7.3 Access pattern for unblocked matrix multiplication

Research on scientific computing led to the design of **blocked** or **tiled algorithms** [320, 606], which improve the **locality of references**. The following is a tiled version of the above algorithm[1] for a block size parameter `B`:

```
for (ii=0; kk<N; ii+=B)
 for (jj=0; jj<N; jj+=B)
   for (kk=0; kk<N; kk+=B)
     for (i=ii; i<min(ii+B-1,N); ii++)
       for (j=jj; j<min(jj+B-1,N); jj++) {
           r=0;
           for (k=kk; k<min(kk+B-1,N); k++)
              r+= X[i][k]*Y[k][j];
           Z[i][j]=r;
       }
```

Now, the two innermost loops are constrained to traverse a block of size B^2 for array `Y`. Suppose that a block of size B^2 fits into the cache. Then, the first execution of the innermost loop will load this block into the cache. During the second execution of the innermost loop, these elements will be reused. Overall, there will be `B-1` reuses of elements of `Y`. Hence, the number of accesses to main memory for elements of this array will be reduced to $N^3/(B-1)$.

∇

Optimizing the reuse factor has been an area of comprehensive research. Initial research focused on the performance improvements that can be obtained by tiling. Performance improvements for matrix multiplication by a factor between 3 and 4.3 were reported by Lam [320]. Improvements increase with an increasing gap between processor and memory speeds. Tiling can also reduce the energy consumption of memory systems [103].

[1] This code was adopted from http://www.netlib.org/utk/papers/autoblock/node2.html.

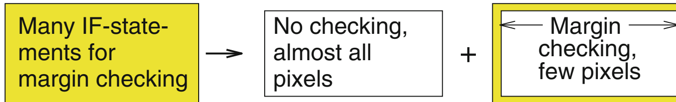

Fig. 7.4 Splitting image processing into regular and special cases

7.1.3 Loop Splitting

Next, we discuss loop splitting as another optimization that can be applied before compilation. Potentially, this optimization could also be added to compilers.

Many image processing algorithms perform some kind of filtering. This filtering consists of considering the information about a certain pixel as well as that of some of its neighbors. Corresponding computations are typically quite regular. However, if the considered pixel is close to the boundary of the image, not all neighboring pixels exist, and the computations must be modified. In a straightforward description of the filtering algorithm, these modifications may result in tests being performed in the innermost loop of the algorithm. A more efficient version of the algorithm can be generated by splitting the loops such that one loop body handles the regular cases and a second loop body handles the exceptions. Figure 7.4 is a graphical representation of this transformation. Margin checking is required for the yellow areas.

Performing this loop splitting manually is very difficult and error-prone. Falk et al. have published an algorithm [159] which also works for larger dimensions automatically. It is based on a sophisticated analysis of accesses to array elements in loops using polyhedral analysis [586]. Optimized solutions are generated using genetic algorithms from the PGAPack library [340]. Falk's algorithm can be implemented, e.g., as a compiler pre-pass tool.

Example 7.5 The following code shows a loop nest from the MPEG-4 standard performing motion estimation:

```
for (z=0; z<20; z++)
 for (x=0; x<36; x++) {x1=4*x;
  for (y=0; y<49; y++) {y1=4*y;
   for (k=0; k<9; k++)  {x2=x1+k-4;
    for (l=0; l<9; l++)  {y2=y1+l-4;
     for (i=0; i<4; i++)  {x3=x1+i; x4=x2+i;
      for (j=0; j<4; j++)  {y3=y1+j; y4=y2+j;
       if (x3<0 || 35<x3 || y3<0 || 48<y3)
        then_block_1; else else_block_1;
       if (x4<0 || 35<x4 || y4<0 || 48<y4)
        then_block_2; else else_block_2;
      }
     }
    }
   }
  }
 }
```

Falk's algorithm detects that the conditions `x3<0` and `y3<0` are never true. The analysis allows transforming the loop nest into the code below. Instead of complex tests in the innermost loop, we have a splitting **`if`**-statement after the third **`for`**-loop statement. Regular cases are handled in the then part of this statement. The **`else`** part handles the relatively small number of remaining cases:

```
    for (z=0; z<20; z++)
      for (x=0; x<36; x++) {x1=4*x;
        for (y=0; y<49; y++)
          if (x>=10 || y>=14)
            for (; y<49; y++)
              for (k=0; k<9; k++)
                for (l=0; l<9; l++ )
                 for (i=0; i<4; i++)
                   for (j=0; j<4; j++) {
                   then_block_1; then_block_2}
          else {y1=4*y;
           for (k=0; k<9; k++) {x2=x1+k-4;
            for (l=0; l<9; l++) {y2=y1+l-4;
             for (i=0; i<4; i++) {x3=x1+i; x4=x2+i;
              for (j=0; j<4; j++) {y3=y1+j; y4=y2+j;
               if ( 0 || 35 <x3 || 0 || 48 < y3)   /* x3<0, y3<0 never true */
                then_block_1; else else_block_1;
               if (x4 < 0|| 35 < x4 || y4 < 0 || 48 < y4)
                then_block_2; else else_block_2;
              }
             }
            }
           }
          }
      }
```

∇

Run-times can be reduced by loop splitting for various applications and architectures. Resulting relative run-times are shown in Fig. 7.5. For the motion estimation algorithm, cycle counts can be reduced by up to about 75% (to 25% of the original value). Substantial savings (larger than for the simple transformations mentioned earlier) are possible.

7.1.4 Array Folding

Some embedded applications, especially in the multimedia domain, include large arrays. Since memory space in embedded systems is limited, options for reducing the storage requirements of arrays should be explored. Figure 7.6 represents the addresses used by five arrays as a function of time. At any particular time, only a subset of array elements is needed. The maximum number of elements needed is called the **address reference window** [122]. In Fig. 7.6, this maximum is indicated by a double-headed arrow. A classical memory allocation for arrays is shown in

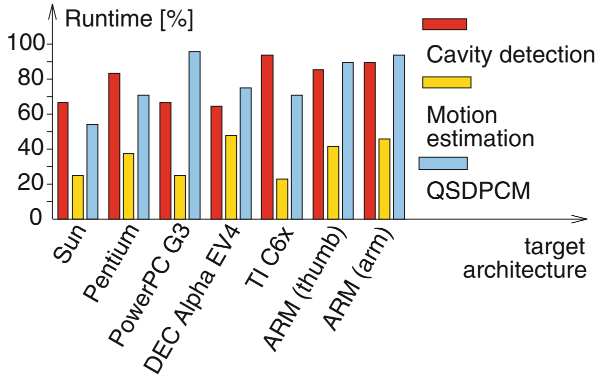

Fig. 7.5 Results for loop splitting

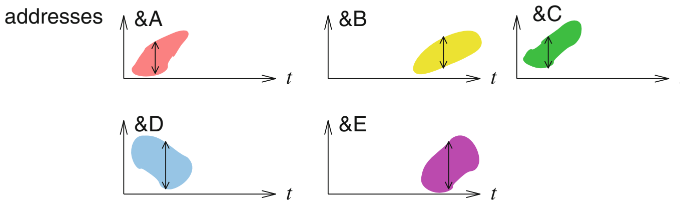

Fig. 7.6 Reference patterns for arrays

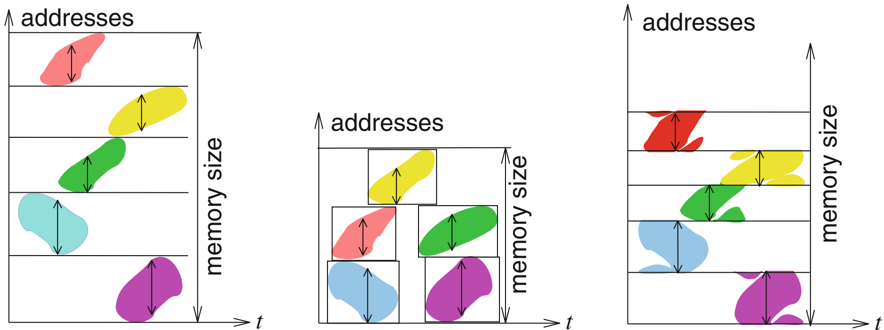

Fig. 7.7 Unfolded (**left**), inter-array folded (**center**), and intra-array folded (**right**) arrays

Fig. 7.7 (left). Each array is allocated the maximum of the space it requires during the entire execution time (if we consider global arrays).

One of the possible improvements, **inter-array folding**, is shown in Fig. 7.7 (center). Arrays which are not needed at overlapping time intervals can share the same memory space. A second improvement, intra-array folding [121], is shown in Fig. 7.7 (right). It takes advantage of the limited sets of components needed

within an array. Storage can be saved at the expense of more complex address computations. The two kinds of foldings can also be combined.

Other forms of high-level transformations have been analyzed by Chung, Benini, and De Micheli [103, 524]. There are many additional contributions in this domain in the compiler community.

7.1.5 Floating-Point to Fixed-Point Conversion

Floating-point to fixed-point conversion is a commonly used optimization technique. This conversion is motivated by the fact that many signal processing standards (such as MPEG-2 or MPEG-4) are specified in the form of C-programs using floating-point data types. It is left to the designer to find an efficient implementation of these standards.

For many signal processing applications, it is possible to replace floating-point numbers with fixed-point numbers (see p. 153). The benefits may be significant. For example, a reduction of the cycle count by 75% and of the energy consumption by 76% has been reported for an MPEG-2 video compression algorithm [225]. However, some loss of precision is normally incurred. More precisely, there is a trade-off between the cost of the implementation and the quality of the algorithm (evaluated, for example, in terms of quality metrics; see Sect. 5.3 on p. 254). For small word lengths, the quality may be seriously affected. Consequently, the quality loss has to be analyzed. This replacement was initially performed manually. However, it is a very tedious and error-prone process.

Therefore, researchers have tried to support this replacement with tools. One of such tools is FRIDGE (*fixed-point programming design environment*) [283, 588]. The functionality of FRIDGE has been made available commercially as part of the Synopsys System Studio tool suite [518].

SystemC can be used for simulating fixed-point data types.

An analysis of the trade-offs between the additional noise introduced and the word length needed was proposed by Shi and Brodersen [486] and also by Menard et al. [390]. The topic continues to attract researchers [334], also in the context of machine learning [454].

7.2 Task-Level Concurrency Management

As mentioned on p. 38, the task graphs' **granularity** is one of their most important properties. Even for hierarchical task graphs, it may be useful to change the granularity of the nodes. The partitioning of specifications into tasks or processes does not necessarily aim at the maximum implementation efficiency. Rather, during the specification phase, a clear separation of concerns and a clean software model are more important than caring about the implementation too much. For example, a clear separation of concerns includes a clear separation of the implementation of abstract data types from their use. As a result of the design process, tasks will

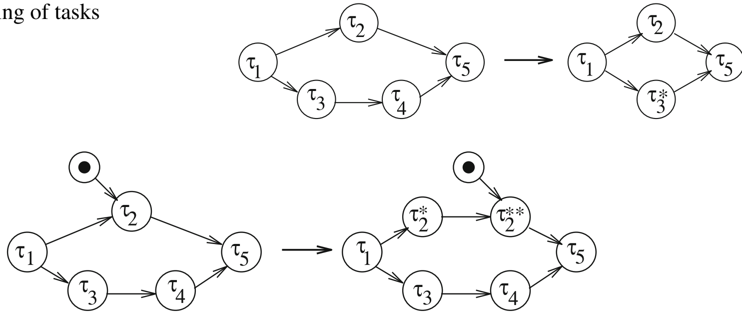

Fig. 7.8 Merging of tasks

Fig. 7.9 Splitting of tasks

typically become objects within the operating system, i.e., processes (cf. Definition 4.1) or threads. Also, we might be using several tasks in a pipelined fashion in our specification, while merging some of them might reduce context switching overhead. Hence, there will not necessarily be a one-to-one correspondence between the tasks in the specification and those in the implementation. This means that a regrouping of tasks may be advisable. Such a regrouping is indeed feasible by merging and splitting of tasks.

Merging of task graphs can be performed whenever some task τ_i is the immediate predecessor of some other task τ_j and if τ_j does not have any other immediate predecessor (see Fig. 7.8 with $\tau_i = \tau_3$ and $\tau_j = \tau_4$). This transformation can lead to a reduced overhead of context switches if the node is implemented in software, and it can lead to a larger potential for optimizations in general.

On the other hand, splitting of tasks may be advantageous, since tasks may be holding resources (like large amounts of memory) while they are waiting for some input. In order to maximize the use of these resources, it may be best to constrain the use of these resources to the time intervals during which these resources are actually needed.

Example 7.6 In Fig. 7.9, we are assuming that task τ_2 requires some input somewhere in its code.

In the initial version, the execution of task τ_2 can only start if this input is available. We can split the node into τ_2^* and τ_2^{**} such that the input is only required for the execution of τ_2^{**}. Now, τ_2^* can start earlier, resulting in more scheduling freedom. This improved scheduling freedom might improve resource utilization and could even enable meeting some deadline. It may also have an impact on the memory required for data storage, since τ_2^* could release some of its memory before terminating and this memory could be used by other tasks while τ_2^{**} is waiting for input. ∇

One might argue that the tasks should release resources like large amounts of memory before waiting for input. However, the readability of the original specification could suffer from caring about implementation issues in an early design phase.

Quite complex transformations of the specifications can be performed with a Petri net-based technique described by Cortadella et al. [111]. Their technique starts with a specification consisting of a set of tasks described in a language called *FlowC*. FlowC extends C with process headers and inter-task communication specified in the form of read and write function calls.

Example 7.7 Figure 7.10 shows an input specification using FlowC. The example uses input ports `IN` and `COEF`, as well as an output port `OUT`. Point-to-point

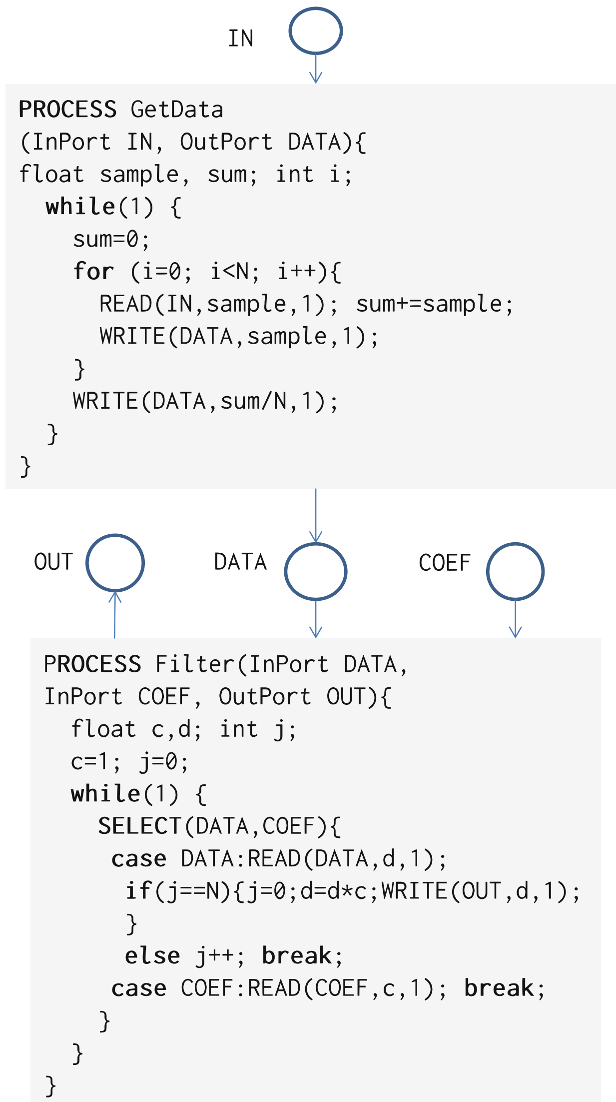

Fig. 7.10 System specification

interprocess communication between processes is realized through a unidirectional buffered channel DATA. Task GetData reads data from the environment and sends it to channel DATA. Each time N samples have been sent, their average value is also sent via the same channel. Task Filter reads N values from the channel (and ignores them), then reads the average value, and multiplies the average value by c. (c can be read from port COEF). Filter writes the result to port OUT. The third parameter in READ and WRITE calls is the number of items to be read or written. READ calls are blocking, and WRITE calls are blocking if the number of items in the channel exceeds a predefined threshold. The SELECT statement has the same semantics as the statement with the same name in Ada (see p. 112): execution of this task is suspended until input arrives from one of the ports. This example meets all criteria for splitting tasks that were mentioned in the context of Fig. 7.9. Both tasks will be waiting for input while occupying resources. Efficiency could be improved by restructuring these tasks. However, the simple splitting of Fig. 7.9 is not sufficient. The technique proposed by Cortadella et al. is more comprehensive: FlowC programs are first translated into (extended) Petri nets. Petri nets for each of the tasks are then merged into a single Petri net. Using results from Petri net theory, new tasks are then generated. Figure 7.11 shows a possible new task structure.

In this new task structure, there is one task which performs all initializations: in addition, there is one task for each of the input ports. An efficient implementation would raise interrupts each time new input is received for a port. There should be a

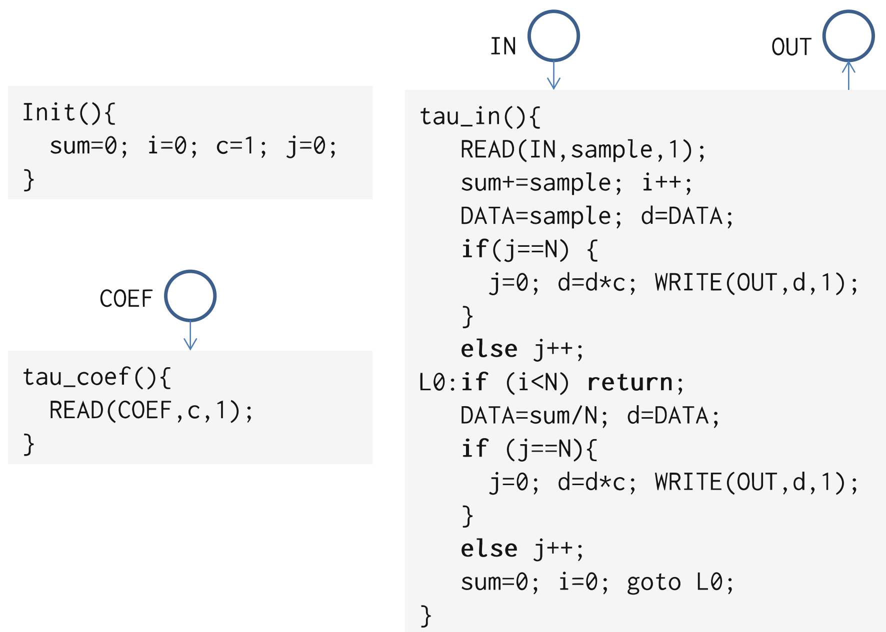

Fig. 7.11 Generated software tasks

unique interrupt per port. The tasks could then be started directly by those interrupts, and there would be no need to invoke the operating system for that. Communication can be implemented as a single shared global variable (assuming a shared address space). The operating system overhead would be small, if required at all.

The code for task `tau_in` shown in Fig. 7.11 is the one that is generated by the Petri net-based inter-task optimization of the task structure. It should be further optimized by intra-task optimizations, since the test performed for the first **`if`** statement is always false (`j` is equal to `i-1` in this case, and `i` and `j` are reset to `0` whenever `i` becomes equal to `N`). For the third **`if`** statement, the test is always true, since this point of control is only reached if `i` is equal to `N` and `i` is equal to `j` whenever label `L0` is reached. Also, the number of variables can be reduced. The following is an optimized version of `tau_in` [111]:

```
tau_in () {
     READ(IN,sample,1);
     sum+=sample;  i++;
     DATA=sample;  d=DATA;              /* merging of DATA & d feasible */
 L0: if (i<N) return;
     DATA=sum/N; d=DATA;
     d=d*c; WRITE(OUT,d,1);
     sum=0; i=0;
     return;
}
```

The optimized version of `tau_in` could be generated by a clever compiler. Hardly any of today's compilers will generate this version, but the example shows the type of transformations required for generating "good" task structures. ∇

For more details about the task generation, refer to Cortadella et al. [111]. Similar optimizations are described in the book by Thoen [538] and in a publication by Meijer et al. [389].

7.3 Compilers for Embedded Systems

7.3.1 Introduction

Obviously, optimizations and compilers are available for the processors used in PCs and servers. Compiler generation for commonly used processors is well understood. For embedded systems, standard compilers are also used in many cases, since they are typically cheap or even freely available.

However, there are several reasons for designing special optimizations and compilers for embedded systems:

- Processor architectures in embedded systems exhibit special features (see p. 143). These features should be exploited by compilers in order to generate efficient

code. Compilation techniques might also have to support compression techniques described on p. 148–p. 150.
- A high efficiency of the code is more important than a high compilation speed.
- Compilers could potentially help to meet and prove real-time constraints. First of all, it would be nice if compilers contained explicit timing models. These could be used for optimizations which really improve the timing behavior. For example, it may be beneficial to freeze certain cache lines in order to prevent frequently executed code from being evicted and reloaded several times.
- Compilers may help to reduce the energy consumption of embedded systems. Compilers performing energy optimizations should be available.
- For embedded systems, there is a larger variety of instruction sets. Hence, there are more processors for which compilers should be available. Sometimes, there is even the request to support the optimization of instruction sets with **retargetable** compilers. For such compilers, the instruction set can be specified as an input to a compiler generation system. Such systems can be used for experimentally modifying instruction sets and then observing the resulting changes for the generated machine code. This is one particular case of **design space exploration** and is supported, for example, by Tensilica tools [82].

Some approaches for retargetable compilers are described in a book on this topic [376]. Optimizations can be found in books by Leupers et al. [337, 338]. In this Section, we will present examples of compilation techniques for embedded processors.

7.3.2 Energy-Aware Compilation

Many embedded systems are mobile systems which must run on batteries. While computational demands on mobile systems are increasing, battery technology is expected to improve only slowly [414]. Hence, the availability of energy is a serious bottleneck for new applications.

Saving energy can be done at various levels, including the fabrication process technology, the device technology, the circuit design, the operating system, and the application algorithms. Adequate translation from algorithms to machine code can also help. High-level optimization techniques such as those presented on p. 349–p. 357 can also help to reduce the energy consumption. In this subsection, we will look at compiler optimizations which can reduce the energy consumption (frequently called low-*power* optimizations). **Energy models** are very essential ingredients of all energy optimizations. Energy models were presented in Chap. 5. Using models like those, the following compiler optimizations have been used for reducing the energy consumption:

- **Energy-aware scheduling:** the order of instructions can be changed as long as the meaning of the program does not change. The order can be changed such that the number of transitions on the instruction bus is minimized. This optimization

can be performed on the output generated by a compiler and therefore does not require any change to the compiler.

- **Energy-aware instruction selection:** typically, there are different instruction sequences for implementing the same source code. In a standard compiler, the number of instructions or the number of cycles is used as a criterion (cost function) for selecting a good sequence. This criterion can be replaced by the energy consumed by that sequence. Steinke and others found that energy-aware instruction selection reduces the energy consumption by some percent [509].
- **Replacing the cost function** is also possible for other standard compiler optimizations, such as register pipelining, loop invariant code motion, etc. Possible improvements are also in the order of a few percent.
- **Exploitation of the memory hierarchy:** as already explained on p. 168, smaller memories provide faster access and consume less energy per access. Therefore, a significant amount of energy can be saved if memory hierarchies are exploited. Of all the compiler optimizations analyzed by Steinke [511, 512], the energy savings enabled by memory hierarchies are the largest. It is therefore beneficial to use small scratchpad memories (SPMs; see p. 172) in addition to large background memories. All accesses to the corresponding address range will then require less energy and are faster than accesses to the larger memory. The compiler should be responsible for allocating variables and instructions to the scratchpad. This approach does, however, require that frequently accessed variables and code sequences are identified and mapped to that address range.

7.3.3 Memory-Architecture Aware Compilation

Compilation Techniques for Scratchpads

The advantages of using SPMs have been clearly demonstrated [36]. Therefore, exploiting SPMs is the most prominent case of memory hierarchy exploitation. Available compilers are usually capable of mapping memory objects to certain address ranges in the memory. Toward this end, the source code typically has to be annotated.

Example 7.8 For ARM® tools, memory segments can be introduced in the source code by using pragmas like

```
# pragma arm section rwdata = "foo", rodata = "bar"
```

Variables declared after this pragma would be mapped to read-write segment `"foo,"` and constants would be mapped to read-only segment `"bar."` Linker commands can then map these segments to particular address ranges, including those belonging to the SPM. ∇

This is the approach taken in compilers for ARM processors [20]. This is not a very comfortable approach, and it would be nice if compilers could perform such a mapping automatically for frequently accessed objects. Therefore, optimization algorithms have been designed. Some of these optimizations have been presented in a separate book [378]. Available SPM optimizations can be classified into two categories:

- **Non-overlaying** (or "static") memory allocation strategies: for these strategies, memory objects will stay in the SPM while the corresponding application is executed.
- **Overlaying** (or "dynamic") memory allocation strategies: for these strategies, memory objects are moved in and out of the SPM at run-time. This is a kind of "compiler-controlled paging," except the migration of objects happens between the SPM and some slower memory and does not involve any disks.

Non-overlaying Allocation

For non-overlaying allocation, we can start by considering the allocation of functions and global variables to the SPM. For this purpose, each function and each global variable can be modeled as a memory object. Let

- S be the size of the SPM,
- sf_i and sv_i be the sizes of function i and variable i, respectively,
- g be the energy consumption saved per access to the SPM (i.e., the difference between the energy required per access to the slow main memory and the one required per access to the SPM),
- nf_i and nv_i be the number of accesses to function i and variable i, respectively,
- xf_i and xv_i be defined as

$$xf_i = \begin{cases} 1 & \text{if function } i \text{ is mapped to the SPM} \\ 0 & \text{otherwise} \end{cases} \tag{7.1}$$

$$xv_i = \begin{cases} 1 & \text{if variable } i \text{ is mapped to the SPM} \\ 0 & \text{otherwise} \end{cases} \tag{7.2}$$

Then, the goal is to maximize the gain

$$G = g\left(\sum_i nf_i * xf_i + \sum_i nv_i * xv_i\right) \tag{7.3}$$

while respecting the size constraint

$$\sum_i sf_i * xf_i + \sum_i sv_i * xv_i \leq S \tag{7.4}$$

The problem is known as a (simple) **knapsack** problem (see p. 320 for the more general case). Standard knapsack algorithms can be used for selecting the objects to be allocated to the SPM. However, Eqs. (7.3) and (7.4) also have the form of an integer linear programming (ILP) problem (see Appendix A), and ILP solvers can be used as well. g is a constant factor in the objective function and is not needed for the solution of the ILP problem. The corresponding optimization can be implemented as a pre-pass optimization (see Fig. 7.12).

The optimization impacts addresses of functions and global variables. Compilers typically allow a manual specification of these addresses in the source code. Hence, no change to the compiler itself is required. The advantage of such a pre-pass optimization is that it can be used with compilers for many different target processors. There is no need to modify a large number of target-specific compilers.

The knapsack model can be extended into various directions:

- **Allocation of basic blocks:** The approach just described only allows the allocation of entire functions or variables to the SPM. As a result, a major fraction of the SPM may remain empty if functions and variables are large. Therefore, we try to reduce the granularity of the objects which are allocated to the SPM. The natural choice is to consider **basic blocks** as memory objects. In addition, we do also consider sets of adjacent basic blocks, where adjacency is defined as being adjacent in the control flow graph [509]. We call such sets of adjacent blocks **multi-basic blocks**. Figure 7.13 shows a control flow graph and the set of considered multi-basic blocks.

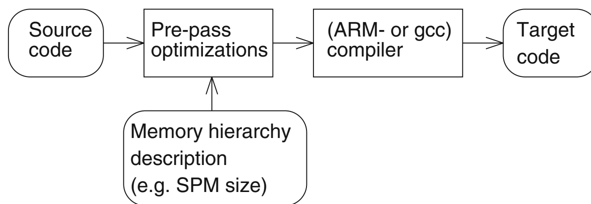

Fig. 7.12 Pre-pass optimization

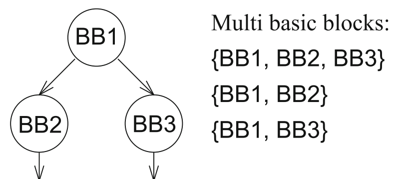

Fig. 7.13 Basic blocks and multi-basic blocks

The ILP model can be extended accordingly. Let

- sb_i and sm_i be the sizes of basic blocks i and multi-basic blocks i, respectively,
- nb_i and nm_i be the number of accesses to basic block i and multi-basic blocks i, respectively,
- xb_i and xm_i be defined as

$$xb_i = \begin{cases} 1 \text{ if basic block } i \text{ is mapped to the SPM} \\ 0 \text{ otherwise} \end{cases} \tag{7.5}$$

$$xm_i = \begin{cases} 1 \text{ if multi basic block } i \text{ is mapped to the SPM} \\ 0 \text{ otherwise} \end{cases} \tag{7.6}$$

Then, the goal is to maximize the gain

$$G = g\left(\sum_i nf_i \cdot xf_i + \sum_i nb_i \cdot xb_i + \sum_i nm_i \cdot xm_i + \sum_i nv_i \cdot xv_i\right) \tag{7.7}$$

while respecting the constraints

$$\sum_i sf_i * xf_i + \sum_i sb_i * xb_i + \sum_i sm_i * xm_i + \sum_i sv_i * xv_i \leq S \tag{7.8}$$

$$\forall \text{ basic blocks } i : xb_i + xf_{fct(i)} + \sum_{i' \in multibasicblock(i)} xm_{i'} \leq 1 \tag{7.9}$$

$fct(i)$ is the function containing basic block i and $multibasicblock(i)$ is the set of multi-basic blocks containing basic block i.

The constraint (7.9) ensures that a basic block is mapped to the SPM only once, instead of potentially being mapped as a member of the enclosing function and a member of a multi-basic block.

Experiments using this model were performed by Steinke et al. [512]. For some benchmark applications, energy reductions of up to about 80% were found, even though the size of the SPM was just a small fraction of the total code size of the application. Results for the bubble sort program are shown in Fig. 7.14. Obviously, larger SPMs lead to a reduced energy consumption in the main memory (see white boxes). The energy required in the CPU is also reduced, since less wait cycles are required. The SPM needs only small amounts of energy (see the tiny blue boxes). Supply voltages have been assumed to be constant, even though a faster execution could have allowed us to scale down frequencies and voltages, leading to an even larger energy reduction.

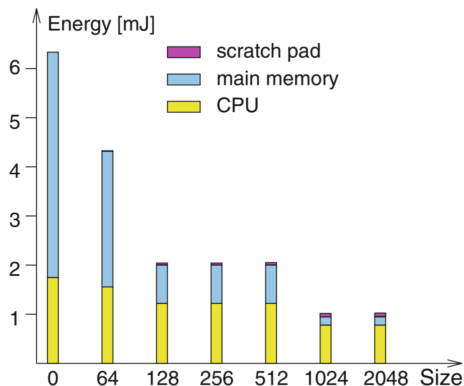

Fig. 7.14 Energy reduction by compiler-based mapping to a SPM

- **Partitioned memories** [572]: Small memories are faster and require less energy per access. Therefore, it makes sense to partition memories into several smaller memories. The ILP model can be extended easily to also model several memories. We do not distinguish between the various types of memory objects (functions, basic blocks, variables, etc.) in this case. An index i represents any memory object. Let
  - S_j be the size of the memory j,
  - s_i be the size of object i (as before),
  - e_j be the energy consumption per access to memory j,
  - n_i the number of accesses to object i (as before),
  - $x_{i,j}$ be defined as

$$x_{i,j} = \begin{cases} 1 & \text{if object } i \text{ is mapped to memory } j \\ 0 & \text{otherwise} \end{cases} \tag{7.10}$$

Instead of maximizing the energy saving, we are now minimizing the overall energy consumption. Hence, the goal is now to minimize

$$C = \sum_j e_j \sum_i x_{i,j} * n_i \tag{7.11}$$

while respecting the constraints

$$\forall j : \sum_i s_i * x_{i,j} \leq S_j \tag{7.12}$$

$$\forall i : \sum_j x_{i,j} = 1 \tag{7.13}$$

Partitioned memories are advantageous especially for varying memory requirements. Storage locations accessed frequently are called the **working set** of an application. Applications with a small working set could use a very small fast memory, whereas applications requiring a larger working set could be allocated to a somewhat larger memory. Therefore, a key advantage of partitioned memories is their ability to adapt to the size of the current working set.

Furthermore, unused memories can be shut down to save additional energy. However, we are considering only the "dynamic" energy consumption caused by accesses to the memory. In addition, there may be some energy consumption even if the memory is idle. This consumption is not considered here. Therefore, savings from shutting down memories are not reflected in Eqs. (7.11) and (7.12).

- **Link/load-time allocation of memory** [420]: Optimizing code at compile time for a certain SPM size has a disadvantage—the code might perform badly if we run it on different variants of some processor if these variants have differently sized SPMs. We would like to avoid requiring different executable files for the different variants of the processor. As a result, we are interested in executables which are independent of the SPM size. This is feasible if we perform the optimization at link time. The proposed approach computes the ratio of the number of accesses divided by the size of a variable at compile time and stores this value together with other information about variables in the executable. At load time, the OS is queried for the size of the SPM. Then, the code is patched such that as many profitable variables as possible are allocated to the SPM.

Overlaying Allocation

Large applications may have multiple hot spots (multiple areas of code containing compute-intensive loops). Non-overlaying approaches fail to provide the best possible results in this context. For such applications, the SPM should be exploited for each of the hot spots. This requires an automatic migration between the layers in the memory hierarchy. For overlaying algorithms, memory objects are migrated between different levels of the hierarchy.[2] This migration can be either programmed explicitly in the application or inserted automatically. Overlaying algorithms are beneficial for applications with multiple hot spots, for which the code or data can be evicting each other. For overlaying algorithms, we are typically assuming that all applications are known at design time such that memory allocation can be considered at this time. Algorithms by Verma [555] and by Udayakumararan et al. [548] are early examples of such algorithms.

Verma's algorithm starts with the CFG of the application to be optimized. For edges of the graph, Verma considers potentially freeing the SPM for locally used

[2]Some of the material in this subsection has also been included in a separate book by the same author and publisher [378].

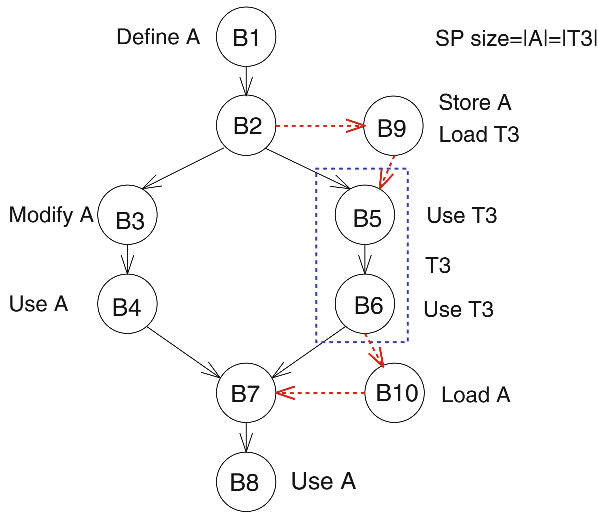

Fig. 7.15 Potential spill code

memory objects by storing these objects in some slower memory and later restoring them. Blocks of code are handled as if they were arrays of data.

Example 7.9 In Fig. 7.15, we are considering control blocks B1–B10 and control flow branching at B2. We assume that array A is defined, modified, and used along the left path. T3 is only used in the right part of the branch. We consider potentially freeing the SPM so that T3 can be locally allocated to the SPM. This requires spill and load operations in potentially inserted blocks B9 and B10 (dotted lines: potential inserts). Cost and benefit of these spill operations are then incorporated into a global ILP. Solving the ILP yields an optimal set of memory copy operations. ∇

For a set of benchmarks, the average reductions in energy consumption and execution time, compared to the non-overlaying case, are 34% and 18%, respectively.

Udayakumararan's algorithm is similar, but it evaluates memory objects according to their number of memory accesses divided by their size. This metric is then used to heuristically guide the optimization process. This approach can also take heap objects into account.

Large arrays are difficult to allocate to SPM. In fact, even a single array can be too large to fit into an SPM. The splitting strategy of Verma [160] is restricted to a single-array splitting. Loop tiling is a more general technique, which can be applied either manually or automatically [344]. Furthermore, array indexes can be analyzed in detail such that frequently accessed array components can be kept in the SPM [357].

Our explanations have so far mainly addressed code and global data. *Stack* and *heap data* require special attention. In both cases, two trivial solutions may be feasible: in some cases, we might prefer not to allocate code or heap data to the SPM at all. In other cases, we could run stack [5] and heap size analysis [219] to check whether stack or heap fit completely into the SPM and, if they do, allocate them to the SPM.

For the heap, Dominguez et al. [134] proposed to analyze the liveness of heap objects. Whenever some heap object is potentially needed, code is generated to ensure that the object will be in the SPM. Objects will always be at the same address, so that the problem of dangling references to heap objects in the SPM is avoided. McIllroy et al. [384] propose a dynamic memory allocator taking characteristics of SPM into account. Bai et al. [33] suggest that the programmer should enclose accesses to global pointers by two functions $p2s$ and $s2p$. These functions provide conversions between global and local (SPM) addresses and also ensure a proper copying of memory contents.

For stack variables, Udayakumararan et al. [548] proposed to use two stacks, one for calls to short functions with their stack being in main memory and one for calls to computationally expensive functions whose stack area is in the SPM. Kannan et al. [281] suggested to keep the top stack frames in the SPM in a circular fashion. During function calls, a check for a sufficient amount of space for the required stack frame is made. If the space is not available, old stack frames are copied to a reserved area in main memory. During returns from function calls, these frames can be copied back. Various optimizations aim at minimizing the necessary checks.

Multiple Threads/Processes

The above approaches are still limited to handling a single process or thread. For multiple threads, moving objects into and out of the SPM at context switch time has to be considered. Verma [556] proposed three different approaches:

1. For the first approach, only a single process owns space in the SPM at any given time. At each context switch, the information of the preempted process in the occupied space is saved, and the information for the process to be executed is restored. This approach is called the **saving/restoring approach**. This approach does not work well with large SPMs, since the copying would consume a significant amount of time and energy.
2. For the second approach, the space in the SPM is partitioned into areas for the various processes. The size of the partitions is determined in a special optimization. The SPM is filled during initialization. No further compiler-controlled copying is required. Therefore, this approach is called the **non-saving approach**. This approach makes sense only for SPMs large enough to contain areas for several processes.

3. The third approach is a **hybrid** approach: The SPM is split into an area jointly used by processes and a second area, in which processes obtain some exclusively allocated space. The size of the two areas is determined in an optimization.

In more dynamic cases, the set of applications may vary during the use of the system. For such cases, dynamic memory managers are appropriate. Pyka [463] published an algorithm based on an SPM manager using indirect addressing and being included in the operating system. This approach also allows the migration of library elements to the SPM. A reduction of the consumed energy of 25%–35% could be achieved despite the additional level of indirect addressing.

This additional level of indirection can be avoided if a memory management unit (see Appendix C) is available. Egger et al. [149] developed a technique exploiting MMUs: at compile time, sections of code are classified as either benefiting or not benefiting from an allocation to the SPM. The code benefiting is stored in a certain area in the virtual address space. Initially, this area is not mapped to physical memory. Therefore, a page fault occurs when the code is accessed for the very first time. Page fault handling then invokes the SPM manager (SPMM) and the SPMM allocates (and deallocates) space in the SPM, always updating the virtual-to-real address translation tables as needed. The approach is designed to handle code and is capable of supporting a dynamically changing set of applications. Unfortunately, the size of current SPMs corresponds to just a few entries in today's page tables, resulting in a coarse-grained SPM allocation.

Supporting Different Architectures and Objectives

We have so far considered different allocation types. Another dimension in SPM allocation is the architectural dimension. Implicitly, we have so far considered single-core systems with a single-memory hierarchy layer and a single SPM. Other architectures exist as well. For example, there may be hybrid systems containing both caches and SPM. We can try to reduce cache misses by selectively allocating SPM space in case of cache conflicts [92, 280, 611]. Also, we can have different memory technologies, like flash memory or other types of nonvolatile RAM [565]. For flash memory, load balancing is important. Also, there might be multiple levels of memories.

SPM can possibly be shared across cores. Also, there may be multiple memory hierarchy levels, some of which can be shared. Liu et al. [349] present an ILP-based approach for this.

Still another dimension in SPM allocation is the objective function. So far, we have focused on energy or run-time minimization. Other objectives can be considered as well. Implicitly, we have modeled the average case energy consumption. We could have modeled the worst case energy consumption (WCEC) instead. The WCEC is an objective considered, for example, by Liu [349]. Reliability and endurance are relevant for the design of reliable applications, in particular in the presence of aging [566]. It may also be necessary to avoid overheating of memories.

7.3.4 Reconciling Compilers and Timing Analysis

Almost all compilers which are available today do not include a timing model. Therefore, the development of real-time software typically has to follow an iterative approach: software is compiled by a compiler which is unaware of any timing information. The resulting code is then analyzed using a timing analyzer such as aiT [4]. If the timing constraints are not met, some of the inputs to the compiler run must be changed, and the procedure has to be repeated. We call this "trial-and-error"-based development of real-time software. This approach suffers from several problems. First of all, the number of required design iterations is initially unknown. Furthermore, the compiler used in this approach is "optimizing," but a precise evaluation of objectives apart from the code size is usually impossible. Hence, compiler writers can only hope that their "optimizations" have a positive impact of the quality of the code in terms of relevant objectives. Due to the complex timing behavior of modern processors, this hope is hardly supported by evidence. Finally, the "trial-and-error"-based development of real-time software requires the designer to find appropriate modifications of the input to the compiler such that the real-time constraints will eventually be met.

This "trial-and-error"-based approach can be avoided if timing analysis is integrated into the compiler. This has been the aim of the development of the worst case execution time-aware compiler (WCC). The development of WCC started at TU Dortmund with an integration of the timing analyzer aiT into an experimental compiler for the TriCore architecture. Figure 7.16 shows the resulting overall structure. WCC uses the ICD-C compiler infrastructure [230] to read and parse C source code. The source is then converted into a "high-level intermediate representation" (HL-IR). The HL-IR is an abstract representation of the source code. Various optimizations can be applied to the HL-IR. The optimized HL-IR is passed to the code selector. The code selector maps source code operations to machine instructions. Machine instructions are represented in the low-level intermediate representation (LLIR). In order to estimate the $WCET_{EST}$, the LLIR is converted into the CRL2 representation used by aiT (using the converter LLIR2CRL). aiT is then able to generate $WCET_{EST}$ for the given machine code. This information is converted back into the LLIR representation (using the converter CRL2LLIR). WCC uses this information to consider $WCET_{EST}$ as the objective function during optimizations. This can be done straightforward for optimizations at the LLIR level. However, many optimizations are performed at the HL-IR-level. $WCET_{EST}$-directed optimizations at this level require using back annotation from the LLIR level to the HL-IR level. ICD-C includes this back annotation.

WCC has been used to study the impact of optimizing for a reduced $WCET_{EST}$ in the compiler. The numerous results include a study of the impact of this objective for register allocation [158]. Results shown in Fig. 7.17 indicate a dramatic impact. $WCET_{EST}$ can be reduced down to 68.8% of the original $WCET_{EST}$ on the average by just using WCET-aware register allocation in WCC. The largest reduction yields a $WCET_{EST}$ of only 24.1% of the original $WCET_{EST}$. The combined effect of

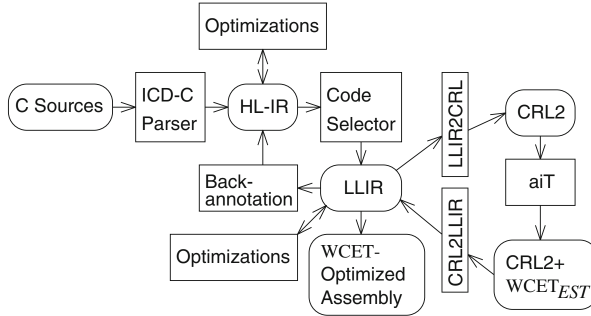

Fig. 7.16 Worst case execution time-aware compiler WCC

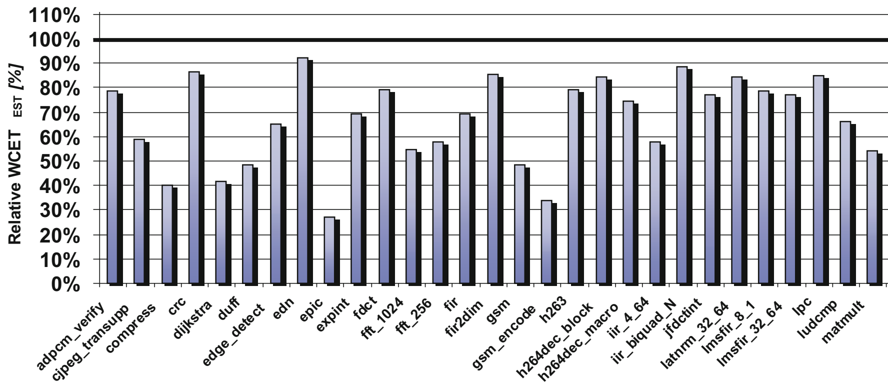

Fig. 7.17 Reduction of $WCET_{EST}$ by WCET-aware register allocation

several such optimizations has been analyzed by Lokuciejewski et al. [353]. For the considered benchmarks, Lokuciejewski found a reduction of down to 57.1% of the original $WCET_{EST}$. Lokuciejewski et al. have also used machine learning to optimize heuristics for WCET reduction [354].

7.4 Power and Thermal Management

7.4.1 Dynamic Voltage and Frequency Scaling (DVFS)

Some embedded processors support dynamic power management (see p. 146) and dynamic voltage scaling (see p. 144). An additional optimization step can be used to exploit these features. Typically, such an optimization step follows code generation by the compiler. Optimizations at this step require a global view of all tasks of the system, including their dependencies, slack times, etc.

Example 7.10 The potential of dynamic voltage scaling is demonstrated in the following example [251]. We assume that we have a processor which runs at three different voltages, 2.5 V, 4.0 V, and 5.0 V. Assuming an energy consumption of 40 nJ per cycle at 5.0 V, Eq. (3.14) can be used to compute the energy consumption at the other voltages (see Table 7.1, where 25 nJ is a rounded value).

Furthermore, we assume that our task needs to execute 10^9 cycles within 25 s. There are several ways of doing this, as can be seen from Figs. 7.18, 7.19, and 7.20. Using the maximum voltage (see Fig. 7.18), it is possible to shut down the processor during the slack time of 5 s (we assume the power consumption to be zero during this time).

Another option is to initially run the processor at full speed and then reduce the voltage when the remaining cycles can be completed at the lowest voltage (see Fig. 7.19).

Finally, we can run the processor at a clock rate just large enough to complete the cycles within the available time (see Fig. 7.20).

The corresponding energy consumptions can be calculated as

$$E_a = 10^9 * 40 * 10^{-9} \text{J} = 40 \text{J} \quad (7.14)$$

$$E_b = 750 * 10^6 * 40 * 10^{-9} + 250 * 10^6 * 10 * 10^{-9} \text{J} = 32.5 \text{J} \quad (7.15)$$

$$E_c = 10^9 * 25 * 10^{-9} \text{J} = 25 \text{J} \quad (7.16)$$

The smallest energy consumption is achieved for the ideal supply voltage of 4 volts, with no idle time at the end. ∇

In the following, we use the term **variable voltage processor** only for processors that allow **any** supply voltage up to a certain maximum. It is expensive to support

Table 7.1 Characteristics of processor with DVFS

V_{dd} [V]	5.0	4.0	2.5
Energy per cycle [nJ]	40	25	10
f_{max} [MHz]	50	40	25
Cycle time [ns]	20	25	40

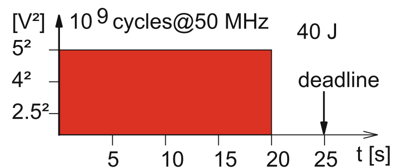

Fig. 7.18 Possible voltage schedule

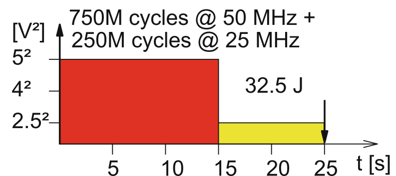

Fig. 7.19 Second voltage schedule

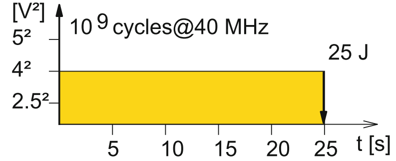

Fig. 7.20 Third voltage schedule

truly variable voltages, and therefore, actual processors support only a few fixed voltages.

The observations made for the above example can be generalized into the following statements. The proofs of these statements are given in the paper by Ishihara and Yasuura [251].

- If a variable voltage processor completes a task before the deadline, the energy consumption can be reduced.[3]
- If a processor uses a single supply voltage V_s and completes a task τ just at its deadline, then V_s is the unique supply voltage which minimizes the energy consumption of τ.

If a processor can only use a number of discrete voltage levels, then a voltage schedule using the two voltages which are the two immediate neighbors of the ideal voltage V_{ideal} can be chosen. These two voltages lead to the minimum energy consumption except if the need to use an integer number of cycles results in a small deviation from the minimum.[4]

The statements can be used for allocating voltages to tasks. Next, we will consider such an allocation. We will use the following notation:

n : the number of tasks
EC_j : the number of executed cycles of task j
L : the number of voltages of the target processor
V_i : the ith voltage, where $1 \leq i \leq L$
f_i : the clock frequency for supply voltage V_i
d : the global deadline at which all tasks must have been completed
SC_j : the average switching capacitance during the execution of task j (SC_j comprises the actual capacitance C_L and the switching activity α (see Eq. (3.14) on page 144))

The voltage scaling problem can then be formulated as an integer linear programming (ILP) problem (see p. 393). Toward this end, we introduce variables $X_{i,j}$ denoting the number of cycles executed at a particular voltage:

$X_{i,j}$: the number of clock cycles task j is executed at voltage V_i

[3]This formulation makes an implicit assumption in lemma 1 of the paper by Ishihara and Yasuura explicit.

[4]This need is not considered in the original paper.

Simplifying assumptions of the ILP model include the following:

- There is one processor that can be operated at a limited number of discrete voltages.
- The time for voltage and frequency switches is negligible.
- The worst case number of cycles for each task is known.

Using these assumptions, the ILP problem can be formulated as follows:
Minimize

$$E = \sum_{j=1}^{n} \sum_{i=1}^{L} SC_j * X_{i,j} * V_i^2 \quad (7.17)$$

subject to

$$\forall j : \sum_{i=1}^{L} X_{i,j} = EC_j \quad (7.18)$$

and

$$\sum_{j=1}^{n} \sum_{i=1}^{L} \frac{X_{i,j}}{f_i} \leq d \quad (7.19)$$

The goal is to find the number $X_{i,j}$ of cycles that each task τ_j is executed at a certain voltage V_i. According to the statements made above, no task will ever need more than two voltages. Using this model, Ishihara and Yasuura show that efficiency is typically improved if tasks have a larger number of voltages to choose from. If large amounts of slack time are available, many voltage levels help to find close to optimal voltage levels. However, four voltage levels do already give good results quite frequently.

There are many cases in which tasks actually run faster than predicted by their worst case execution times. This cannot be exploited by the above algorithm. This limitation can be removed by using checkpoints at which actual and worst case execution times are compared and then to use this information to potentially scale down the voltage [30]. Also, voltage scaling in multi-rate task graphs was proposed [479]. DVFS can be combined with other optimizations such as body biasing [369]. Body biasing is a technique for reducing leakage currents.

7.4.2 Dynamic Power Management (DPM)

In order to reduce the energy consumption, we can also take advantage of power-saving states, as introduced on p. 146. The essential question for exploiting DPM is:

when should we go to a power-saving state? Straightforward approaches just use a simple timer to transition into a power-saving state. More sophisticated approaches model the idle times by stochastic processes and use these to predict the use of subsystems with more accuracy. Models based on exponential distributions have been shown to be inaccurate. Sufficiently accurate models include those based on renewal theory [490].

A comprehensive discussion of power management was published (see, for example, [46, 356]). There are also advanced algorithms which integrate DVS and DPM into a single optimization approach for saving energy [491].

Allocating voltages and computing transition times for DPM may be two of the last steps of optimizing embedded software.

Power management is also linked to thermal management.

7.4.3 Thermal Management

Design time planning of the thermal behavior would need to leave large margins in terms of available performance. Hence, it is necessary to use run-time monitoring of temperatures. This means that thermal sensors must be available in systems which potentially could get too hot. This information is then used to control the generation of additional heat and possibly has an impact on cooling mechanisms as well. Many users of mobile phones may already have observed this: it is, for example, very common to stop charging a mobile phone when it is already too hot. Controlling fans (when available) can be considered as another case of thermal management. Also, systems may be shutting down partially or completely, if temperatures are exceeding maximum thresholds. Shutdown areas of silicon chips can be called "dark silicon." Some systems may be reducing the clock frequencies and voltages. There are also other options like a reduction of the performance by intentionally not using some of the available hardware. It is possible, for example, to issue less instructions per clock cycle or not to use some of the processor pipelines. For multiprocessor systems, tasks may be automatically migrated between various processors. In all of these cases, the objective "temperature" is evaluated at run-time and used to have an impact at run-time. Avoiding overheating is the goal of the work reported by Merkel et al. [391] and by Donald et al. [135]. Using temperature sensors to control the system means that control loops are being created. Potentially, such loops could start to oscillate. Atienza et al. have compared the behavior of various control strategies and came to the conclusion that an advanced control loop algorithm provides the best results, with a higher computing performance at a lower temperature, compared to standard approaches [610]. The details of this control loop design would be beyond the scope of a textbook useful for undergraduate students.

7.5 Problems

We suggest solving the following problems either at home or during a flipped classroom session:

7.1 Loop unrolling is one of the potentially useful optimizations. Please name two potential benefits and two potential problems!

7.2 We assume that you want to use loop tiling. How can you adjust the tiling to the memory architecture at hand?

7.3 For which architectures would you expect the largest benefits from a replacement of floating-point arithmetic by fixed-point arithmetic?

7.4 Provide an overview over techniques for taking advantage of scratch pad memories!

7.5 Consider the following program:

```
 1  #include <stdio.h>
 2  #define DATALEN 15
 3  #define FILTERTAPS 5
 4  double x[DATALEN] = { 128.0, 130.0, 180.0, 140.0, 120.0,
 5                       110.0, 107.0, 103.5, 102.0,  90.0,
 6                        84.0,  70.0,  30.0,  77.3,  95.7 };
 7  const double h[FILTERTAPS]={0.125,-0.25,0.5,-0.25,0.125};
 8  double y[DATALEN]; // result;
 9  int main(void) {
10    int i,n;
11    for(i=0;i<DATALEN;++i) {
12      y[i] = 0;
13      for(n=0; n < FILTERTAPS; ++n)
14        if ((i-n) > = 0) y[i] += h[n]*x[i-n];
15    }
16    for(i = 0; i < DATALEN; ++i) printf("%.2f ",y[i]);
17    return 0;
18  }
```

Perform at least the following optimizations:

- Removal of the `if` in the innermost loop (line 14)
- Loop unrolling (line 13)
- Constant propagation
- Floating-point to fixed-point conversion
- Avoidance of all accesses to arrays

Please provide the optimized version of the program after each of the transformations and do also check for consistent results!

Table 7.2 SPM mapping: **left**, accesses to variables; **right**, memory characteristics

Variable	Size [bytes]	Number of accesses
a	1024	16
b	2048	1024
c	512	2048
d	256	512
e	128	256
f	1024	512
g	512	64
h	256	512

Memory	Size [bytes]	Energy per access
Scratchpad	4096 (4 k)	1.3 nJ
Main memory	262,144 (256 k)	31 nJ

7.6 Suppose that your computer is equipped with a main memory and a scratchpad memory. Sizes and the required energy per access are shown in Table 7.2 (right). Characteristics of accesses to variables are as indicated in Table 7.2 (left).

Which of those variables should be allocated to the scratchpad memory, provided that we use a static, non-overlaying allocation of variables? Use the integer linear problem (ILP) model to select the variables. Your result should include the ILP model as well as the results. You may use the *lp_solve* program [17] to solve your ILP problem.

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.