Memory-Aware Optimization of Embedded Software for Multiple Objectives

26

Peter Marwedel, Heiko Falk, and Olaf Neugebauer

Abstract

Information processing in Cyber-Physical Systems (CPSs) has to respect a variety of constraints and objectives such as response and execution time, energy consumption, Quality of Service (QoS), size, and cost. Due to the large impact of the size of memories on their energy consumption and access times, an exploitation of memory characteristics offers a large potential for optimizations. In this chapter, we will describe optimization approaches proposed by our research groups. We will start with optimizations for single objectives, such as energy consumption and execution time. As a consequence of considering hard real-time systems, special attention is on the minimization of the Worst-Case Execution Time (WCET) within compilers. Three WCET reduction techniques are analyzed: exploitation of scratchpads, instruction cache locking, and cache partitioning for multitask systems. The last section presents an approach for considering trade-offs between multiple objectives in the design of a cyber-physical sensor system for the detection of bio-viruses.

Acronyms

CFG	Control-Flow Graph
CPS	Cyber-Physical System
CPU	Central Processing Unit
CRPD	Cache-Related Preemption Delay
DRAM	Dynamic Random-Access Memory

P. Marwedel (✉) • O. Neugebauer
Computer Science, TU Dortmund University, Dortmund, Germany
e-mail: Peter.Marwedel@tu-dortmund.de; Olaf.Neugebauer@tu-dortmund.de

H. Falk
Institute of Embedded Systems, Hamburg University of Technology, Hamburg, Germany
e-mail: Heiko.Falk@tuhh.de

© Springer Science+Business Media Dordrecht 2017
S. Ha, J. Teich (eds.), *Handbook of Hardware/Software Codesign*,
DOI 10.1007/978-94-017-7267-9_27

FIFO	First-In First-Out
GA	Genetic Algorithm
GPU	Graphics Processing Unit
ILP	Integer Linear Program
LRU	Least-Recently Used
MMU	Memory Management Unit
PAMONO	Plasmon-Assisted Microscopy of Nano-Objects
QoS	Quality of Service
SPM	Scratchpad Memory
SRAM	Static Random-Access Memory
SVM	Support Vector Machine
WCC	WCET-aware C Compiler
WCEC	Worst-Case Energy Consumption
WCEP	Worst-Case Execution Path
WCET	Worst-Case Execution Time

Contents

26.1 Introduction

This chapter considers the mapping of software applications to execution platforms for embedded systems. Embedded systems are information processing systems embedded into enclosing products such as cars or smart homes [29]. In combination with their physical environment, embedded systems form so-called Cyber-Physical Systems (CPSs). According to the National Science Foundation (NSF), "*CPSs are engineered systems that are built from and depend upon the synergy of computational and physical components*" [31]. In our view, embedded systems can be seen as the information processing part in a CPS. Due to the integration with the physical environment, embedded systems have to meet a large set of functional requirements, constraints, and objectives. Hence, in addition to meeting the functional requirements, optimization for the relevant objectives within the design space imposed by the constraints is an essential part of design methodologies for embedded systems. Analyzing currently available technology, it turns out that much of the potential for optimizations concerns memories and their usage. In the following sections, the existence of this potential will be proved by means of examples. The examples are intended to provide an overview over optimization potential in this area, using our research results for demonstration. Specific pointers to our publications are included for further reference and more in-depth discussion.

26.2 Constraints and Objectives

One of the characteristics of embedded systems is the need to consider a large variety of constraints and objectives during their design.

26.2.1 Timing

Embedded systems often have to meet real-time constraints that make them real-time systems. Not completing computations within a given time can result in a serious loss of the quality provided by the system (e.g., if the audio or video quality is affected) or may cause harm to the user (e.g., if cars, trains, or planes do not operate in the predicted way). Time constraints are called *hard* if not meeting them could result in a catastrophe. All other time constraints are called *soft*.

During the design of real-time systems, the Worst-Case Execution Time (WCET) plays an important role. The WCET is the largest execution time of a program for any input and any initial execution state of the hardware platform. In general, it is undecidable whether or not the WCET is finite, because it is undecidable whether or not a program terminates. Hence, the WCET can only be computed for certain simply structured programs. For realistic and general programs, it is usually practically impossible to compute the WCET. Instead, reliable upper bounds have to be determined by sound methods. Such upper bounds are usually called estimated WCET (WCET_{EST}) values and should have at least two properties:

1. The bounds should be safe ($\text{WCET}_{EST} \geq \text{WCET}$).
2. The bounds should be tight ($\text{WCET}_{EST} - \text{WCET} \rightsquigarrow 0$).

If safe WCET guarantees for hard real-time systems are needed, static program analyses are used. At binary code level, such static analyzers estimate register values in order to identify loop counters, determine loop iteration counts, and extract hardware-specific states of a processor's caches and pipelines. The path analysis stage finally estimates a program's global WCET by finding that path within a program's Control-Flow Graph (CFG) that has the maximal WCET – the so-called Worst-Case Execution Path (WCEP). The length of this longest path is the sum of the products $T * C$ over all blocks along the path, where T denotes a block's maximum execution time and C represents the block's maximal execution count.

26.2.2 Energy Consumption and Thermal Behavior

These days, we are almost exclusively using electrical devices to process information. Unfortunately, the operation of known devices requires the conversion of electrical energy into thermal energy. There are various reasons for trying to keep the amount of dissipated electrical energy as small as possible. For example, we would like to keep the impact on global warming as small as possible and we would like to avoid high operating temperatures and too high current densities. Energy may be available only in limited quantities. For mobile systems, electrical energy has to be either carried around with the system (e.g., in the form of batteries) or harvested (e.g., by using solar cells).

Using the consumed energy as an objective or constraint is not easy, since the amount of consumed energy depends on many factors. There are essentially two ways of estimating this objective: estimation can be either based on measurements for real hardware or based on computer models. Measurements can provide very precise results but can be performed only for existing hardware. Models can be used also for non-existing hardware, but they are inherently less precise.

The thermal behavior is very much linked to the energy consumption: the conversion of electrical energy into thermal energy is a source of heating the system. Thermal modeling has to take the thermal resistance between the system and the environment as well as thermal capacities into account. Again, computer models as well as measurements can be used.

26.2.3 Quality of Service and Precision

Overall, embedded systems have to provide some service, e.g., controlling a physical behavior (such as braking a car), showing some video, or generating some functional information. Such a service can be of high quality or of a reduced quality. For example, a video can have various signal to noise ratios and different timing jitters. Control loops may be needing different amounts of time to stabilize. For functional information, there may be a deviation between known precise results and a computed approximation. In the following, the different levels of service will be called Quality of Service (QoS), and we will consider the precision of some functional result as a special case.

26.2.4 Safety, Security, and Dependability

Embedded systems may have a direct impact on their physical environment. Therefore, if embedded systems fail to perform the intended service, the physical environment may be at risk. System failures can be caused by internal malfunctions of the system as well as by attackers compromising the system. As a result, safety and security of embedded systems are extremely important.

Due to the impact on the physical environment, dependability of embedded systems is also important. By dependability, we capture the fact that an initially correctly designed and manufactured system may fail due to some internal fault, e.g., a bit flip in memory. Various physical effects can lead to such faults. Shrinking dimensions of microelectronic circuits are known to increase the rate of such faults [6]. Hence, they will have to be considered more carefully in the future.

26.2.5 Further Constraints and Objectives

There are many more constraints and objectives which are relevant. These include size, cost, and weight or the availability of hardware platforms. Embedded systems may have to resist certain types of radiation and may need to be environmentally friendly disposable. Not all of these can be described in detail in this chapter.

The principle of Pareto optimality can be used to take design decisions in the presence of multiple objectives. For further information on multi-objective design space exploitation, refer to our textbook [29] and to "▸ Chap. 6, "Optimization Strategies in Design Space Exploration"" of this book.

26.3 Optimization Potential in the Memory System

Much optimization potential is available in the memory system, because small memories are faster and consume less energy per access than larger memories. This observation was already made very early by Burks, Goldstein, and von Neumann in 1946 [7]:

> Ideally one would desire an indefinitely large memory capacity such that any particular ... word ... would be immediately available – i.e. in a time which is ... shorter than the operation time of a fast electronic multiplier. ... It does not seem possible physically to achieve such a capacity. We are therefore forced to recognize the possibility of constructing a hierarchy of memories, each of which has greater capacity than the preceding but which is less quickly accessible.

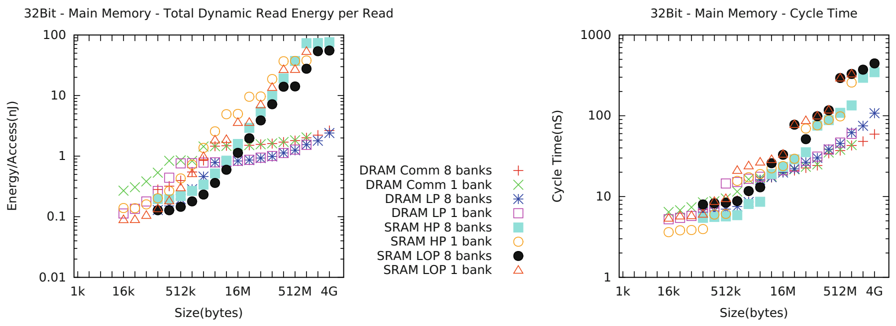

Fig. 26.1 Access times and energy consumption for DRAM and for SRAM

Figure 26.1 shows access times and energy consumptions per access for contemporary Dynamic Random-Access Memories (DRAMs) as well as for Static Random-Access Memories (SRAMs), for high-power and for low-power variants of these and for a single and eight banks. Numbers have been computed with CACTI [19]. Access times as well as energy consumptions vary by more than two orders of magnitude. Due to this, memory hierarchies have been introduced. Their key goal is to assign frequently accessed information to small and fast layers of the hierarchy such that, overall, the impression of a fast, energy-efficient, and still large memory is achieved on the average.

Small and fast memories thus act as buffers between main memory and the processor. For embedded systems, the architecture of these small memories has to be highly energy-efficient and must guarantee a predictable real-time performance. (More information on power and energy models can be found in ▶ Chap. 27, "Microarchitecture-Level SoC Design" of this book).

26.3.1 Caches

Let us briefly look again at some of the memories which were described in detail in ▶ Chap. 13, "Memory Architectures" of this book. *Cache*-based memory hierarchies are today's state of the art, because caches are fully transparent to the software running on a system. No code modification has to be done, since caches are hardware controlled. Caches are effective in exploiting *temporal locality* and *spatial locality*. The former means that particular memory locations will be accessed multiple times within a short period of time. The latter refers to the reference of contiguous memory locations over time.

N-way set-associative caches are organized as a matrix with N columns (usually called *ways*). During a memory access with a given address, the least significant bits of this address (i.e., its index bits) unambiguously identify the row of the cache matrix (usually called *set*) that potentially buffers the requested memory cell's contents. Within the selected cache set, the requested item can now reside in any of the N ways. Thus, the most significant address bits (the tag bits) are compared with the tag bits buffered in all N ways of the selected set. If the tag bits match (cache hit), the requested memory cell is buffered in the cache and the data buffered in the identified way and set is returned to the processor. If no tag comparison matches (cache miss), the cache does not buffer the requested memory cell. A *replacement policy* is responsible for deciding which item to evict from the currently selected set if all ways of that set are currently occupied, but a new item shall be inserted in this set.

This architecture of set-associative caches combined with replacement policies enables a very high flexibility of caches so that they can autonomously adapt to varying memory access patterns issued by the processor. However, the drawbacks of caches are their large penalties in terms of the objectives introduced in Sect. 26.2. Caches exhibit a rather high-energy dissipation due to the additional memory required to store the tag bits and due to the hardware comparators performing the tag bit comparison for the currently selected cache set. Regarding real-time deadlines, caches are notorious for their inherent unpredictability. Depending on its replacement policy, it is hard, if not impossible, to predict during a static WCET analysis if a memory access results in a definite cache hit or miss. If a static WCET analyzer is uncertain about the cache's behavior, it has to assume the worst-case behavior of the cache which frequently leads to highly overestimated WCET_{EST} values.

Modern architectures support *cache locking*, i.e., cache cells are protected from being evicted by effectively partially disabling the replacement policy. This way, it is possible to predict access times of data or instructions that have been locked in the cache and to make precise statements about the cache's worst-case timing.

26.3.2 Scratchpad Memories

As an alternative to caches, small and "conventional" memories can be mapped into the processor's address space. These memories are frequently called Scratchpad Memories (SPMs) and differ from caches in that they are not operating autonomously in hardware. Instead, a simple address decoder decides whether a memory cell that is accessed by the processor is part of the SPM's address space or not, and the requested item is then fetched from the SPM or from some other memory.

Since SPMs completely lack tag memories and comparators, their energy efficiency is significantly higher than that of caches [5]. Furthermore, an access to the SPM always takes a constant time which is usually one clock cycle. As a consequence, varying memory access latencies due to cache misses or hits cannot occur in SPM-based architectures, thus rendering WCET estimates extremely tight and accurate. The drawback of SPMs is their lacking flexibility. Since they are unable to decide autonomously in hardware which items to buffer in and to evict from the SPM memory, there must be some software instance that assigns energy- or timing-critical parts of a program's code or data to the SPM. Frequently, this instance is the compiler that applies *scratchpad allocation techniques* and that determines a memory layout of a program such that it exploits the available SPM resources best. See ▶ Chap. 13, "Memory Architectures" of this book for a detailed comparison of caches and SPMs.

26.3.3 A Bound for Improvements

By how much can we improve memory references with respect to some objective on the average? Suppose that we are given two layers of the memory hierarchy and that memory m_i is closer to the processor and memory m_{i+1} further away from the processor. Suppose that a_i is the access time of memory m_i and a_{i+1} is the access time of m_{i+1}. Furthermore, let us assume that a fraction P of memory references to m_{i+1} can be replaced by references to m_i, leaving a fraction of $(1 - P)$ (the miss rate) of the memory references using m_{i+1}. Then, the average access time is

$$\text{average new access time} = P \cdot a_i + (1 - P) \cdot a_{i+1} \tag{26.1}$$

Let S be the ratio of access times (for available memory technologies, S can easily be in the order of 100):

$$S = \frac{a_{i+1}}{a_i} \tag{26.2}$$

The relative saving is

$$\begin{aligned}
\text{relative saving} &= \frac{\text{average old access time} - \text{average new access time}}{\text{average old access time}} \\
&= \frac{a_{i+1} - P \cdot a_i - a_{i+1} + P \cdot a_{i+1}}{a_{i+1}} \\
&= \frac{a_i \cdot S - P \cdot a_i - a_i \cdot S + P \cdot a_i \cdot S}{a_i \cdot S} \\
&= \frac{S - P - S + P \cdot S}{S} \\
&= P - \frac{P}{S}
\end{aligned} \tag{26.3}$$

The speedup of memory accesses can then be computed as follows:

$$\begin{aligned}
\text{speedup} &= \frac{\text{average old access time}}{\text{average new access time}} \\
&= \frac{a_{i+1}}{P \cdot a_i + (1 - P) \cdot a_{i+1}} \\
&= \frac{a_{i+1}}{P \cdot \frac{a_{i+1}}{S} + (1 - P) \cdot a_{i+1}} \\
&= \frac{1}{\frac{P}{S} + (1 - P)}
\end{aligned} \tag{26.4}$$

If S gets large, the first term in the denominator approaches zero and the improvement gets limited by the second term, $(1 - P)$. Hence, making S very large has a limited benefit when the miss rate cannot be made smaller. This effect is well known for caches. In practice, this means that the miss rate must be made as small as possible. Due to the problem of making the miss rate small, we have to make sure that almost every memory object can potentially be mapped to faster levels in the memory hierarchy. Excluding the stack, heap, or other memory objects from a mapping to fast memories would have a negative impact of the feasible speedup. In general, the goal of an ideal memory hierarchy is not always reached. Drepper has shown for the case of single-core systems that run times of programs can change by orders of magnitude if their working set exceeds the sizes of caches [13]. Hence, a clever use of memory hierarchies is already needed for single cores.

Equation (26.4) also corresponds to Amdahl's law [3], describing bounds for speedup by parallelization, where P is the fraction of the code which is parallelized and S is the speedup during parallel execution.

Equations (26.1) to (26.4) can also be generalized to capture effects for objectives other than access times. In particular, they can be applied to the case of modeling access energies and the resulting improvements.

26.3.4 Importance of Memory-Aware Load Balancing

For multi-core systems, an excessive amount of threads can lead to a lack of memory, as demonstrated, e.g., by Kotthaus and Korb [23]. Figure 26.2 (resulting

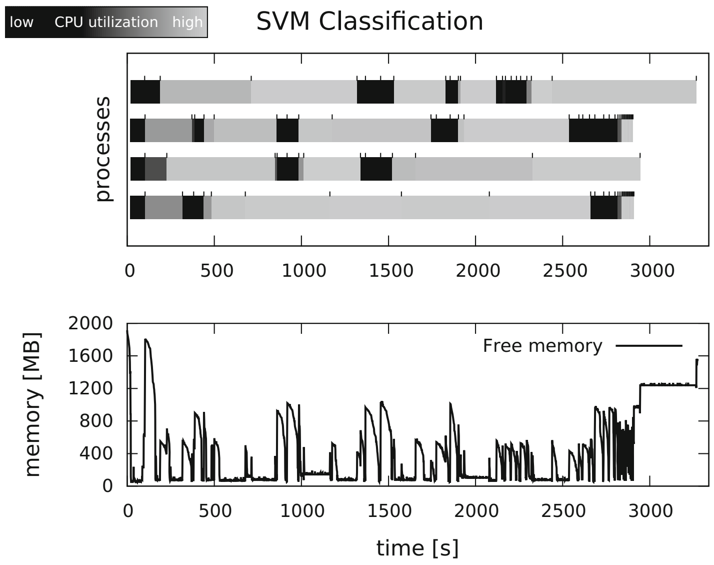

Fig. 26.2 SVM application on a four-core system: lack of free memory resulting in idling cores

from the experiments of Kottaus and Korb) shows profiling results for a Support Vector Machine (SVM) application running on a four-core system.

The application is programmed in the R language and is executed by version 3 of the R system running on a Lenovo L512 comprising an i5 processor. Due to memory-unaware allocation of cores, there are phases in which the system runs out of free memory. This happens even though load balancing of R is turned on. However, R is unaware of actual resource requirements. As a result, there is sometimes no free memory remaining. This is indicated by horizontal lines in the lower part of the diagram. Due to a too large number of processes, the system runs out of main memory and suffers from an increased swapping activity. This example demonstrates that memory-unaware allocation of computing resources results in wasting resources and cannot be efficient. We should care about required memory resources even for scenarios in which we address the programming of multi-core systems at a high level. Therefore, we will be looking at memory allocation in more detail in the remaining sections of this chapter.

26.4 Scratchpad Allocation Algorithms

26.4.1 Classification

In an earlier paper [5], we provided a detailed side-by-side comparison of caches and SPMs with respect to access times, energy consumptions, and silicon areas. A detailed comparison is also included in ▸ Chap. 13, "Memory Architectures" of this book.

In contrast to caches, SPMs must be explicitly managed by software. In the following, we classify the approaches to SPM management according to three dimensions:

- The type of allocation algorithm
- The type of architecture
- The optimization objective

We start by looking at allocation algorithms. SPM allocation algorithms can be classified into non-overlaying (or "static") and overlaying (or "dynamic") algorithms. For the first type of algorithms, memory objects are resident in the SPM during the entire lifetime of an application, whereas for the latter, objects are moved between the memories during run time.

26.4.2 Non-overlaying Allocation Algorithms

For the non-overlaying case, the optimization problem for energy or run-time optimization can be modeled as a Knapsack problem or as an Integer Linear Program (ILP).

Let i denote a memory object and let s_i denote its size. Let Δ_i denote the **saving** with respect to the considered objective if i is mapped to the SPM. The saving is the difference between the objective values for a mapping to some main memory and the SPM. Let S_{SPM} denote the size of the SPM. Let x_i be 1 if i is mapped to the SPM and 0 otherwise. Then, the following ILP model can be used to find an optimal mapping of memory objects to the SPM:

$$\text{Maximize} \quad \sum_i x_i * \Delta_i \tag{26.5}$$

$$\text{Subject to}$$

$$\sum_i x_i * s_i \leq S_{SPM} \tag{26.6}$$

Algorithms by Steinke [38] and by Verma [44] are examples based on such models. They are particular examples of hardware-aware compilation discussed in ▶ Chap. 25, "Hardware-Aware Compilation" of this book. In order to minimize the fraction $(1 - P)$ of "unimproved" memory references, most of our optimizations consider code and data references. For data references, global data can be easily taken into account. We have also considered stack variables. As a result, large savings (as computed by Equation (26.3)) have been observed.

We will demonstrate these for partitioned memories. Here, we have at our disposal J memories, each of them having an energy consumption e_j per access and a size S_j. Let n_i be the number of accesses to memory object i. A decision variable $x_{i,j}$ will be 1 if memory object i is mapped to memory j and 0 otherwise. Then, the following ILP model allows us to minimize the energy consumption:

$$\text{Minimize} \quad \sum_j e_j * \left(\sum_i x_{i,j} * n_i \right) \tag{26.7}$$

$$\text{Subject to}$$

$$\forall j \in J : \sum_i x_{i,j} * s_i \leq S_j \tag{26.8}$$

$$\forall i : \sum_j x_{i,j} = 1 \tag{26.9}$$

Figure 26.3 shows results for partitioned SPMs with 1 to 8 partitions.

For a single SPM, the savings (as computed by Equation (26.3) but indicated as a percentage, rather than a fraction) decrease when the SPM is larger than the working set of the application. For partitioned SPMs, the saving remains at the maximum, even for oversized SPMs. These savings refer to dynamic power consumption. Partitioned SPMs provide even larger advantages when leakage power is also taken into account.

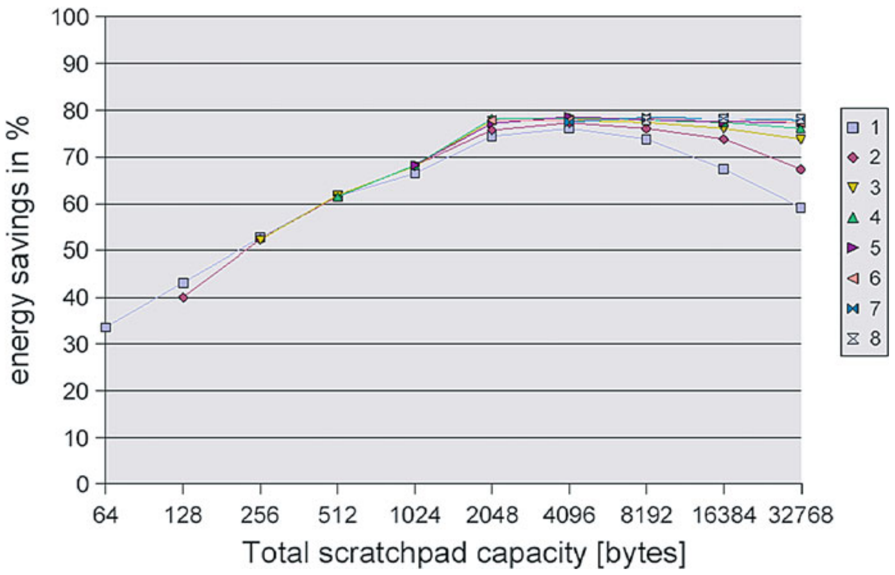

Fig. 26.3 Energy savings achieved by SPM allocation of a GSM application

26.4.3 Overlaying Allocation Algorithms

For overlaying algorithms, memory objects are migrated between different levels of the hierarchy. This migration can be either explicitly programmed in the application or inserted automatically. Overlaying algorithms are beneficial for applications with multiple hotspots, for which the code 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 [44] and Udayakumararan et al. [41] 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 memory objects.

In Fig. 26.4, we are considering control blocks B1 to 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 (thin and 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. Blocks of code are handled as if they were arrays of data.

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

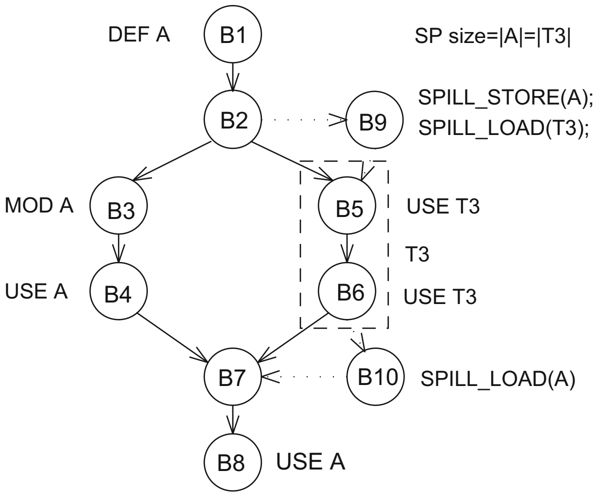

Fig. 26.4 Potential spill code

used to heuristically guide the optimization process. This approach can also take heap objects into account.

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 [36] published an algorithm based on an SPM manager which is part of the operating system.

Egger et al. [9] proposed to exploit an existing Memory Management Unit (MMU) for dynamic replacements within the SPM. In his approach, code objects are classified into those that should potentially be moved into the SPM and those that should not. Potential SPM objects are then grouped into pages. Corresponding MMU entries are initially set to invalid. During execution, MMU exceptions are generated for accesses to SPM candidates not (yet) available in SPM. An exception handler is then invoked. The handler decides which memory objects to move into the SPM and which objects to move out. 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.

Large arrays are difficult to allocate to SPMs. In fact, even a single array can be too large to fit into an SPM. The splitting strategy of Verma [16] is restricted to a single-array splitting. Loop tiling is a more general technique, which can be applied either manually or automatically [24]. Furthermore, array indexes can be analyzed in detail such that frequently accessed array components can be kept in the SPM [27].

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. Obviously, this would have an immediate effect on the bound for the achievable speedup as per Equation (26.4). In other cases, we could run stack [2] and heap size analysis [18] 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. [12] 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. [30] propose a dynamic memory allocator taking characteristics of SPM into account. Bai et al. [4] 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 the stack, Udayakumararan et al. [41] 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. [22] 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.

26.4.4 Supporting Different Architectures and Objectives

A second dimension in SPM allocation (in addition to the allocation type) 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 SPMs. We can try to reduce cache misses by selectively allocating SPM space in case of cache conflicts [8, 21, 48]. Also, we can have different memory technologies, like flash memory or other types of non-volatile RAM [45]. For flash memory, load balancing is important. Also, there might be multiple levels of memories. So far, we have just considered single-core processors. For multi-core systems, new tasks and options exist. SPMs can possibly be shared across cores. Also, there may be multiple memory hierarchy levels, some of which can be shared. Liu et al. [25] present an ILP-based approach for this.

A third 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. The Worst-Case Energy Consumption (WCEC) is an objective considered, for example, by Liu [25]. Reliability and endurance are relevant for the design of reliable applications, in particular in the presence of aging [46]. It may also be necessary to avoid overheating of memories. From among other possible objectives, we will be looking at the WCET in the following sections.

26.5 WCET-Oriented Compiler Strategies

In contrast to simple optimization objectives like, e.g., energy consumption that can be modeled using single values like, e.g., Δ_i or e_j (cf. Sect. 26.4.2), the systematic reduction of WCET estimates is much more subtle due to the nature of the WCET. As already motivated in Sect. 26.2.1, the WCET of a program corresponds to the length of the longest path WCEP through a program's CFG. Thus, WCET-oriented optimizations must exclusively focus on those parts of the program that lie on the WCEP. The optimization of parts of the program aside the WCEP is ineffective, since this does not shorten the WCEP. Therefore, optimization strategies for WCET reduction must have detailed knowledge about the WCEP.

Unfortunately, this WCEP can be highly unstable in the course of an optimization. Consider the CFG of a function `main` in Fig. 26.5a, consisting of five basic blocks each of them having the indicated WCET values given in clock cycles. Obviously, the longest path through this CFG is `main`, `a`, `b`, and `c`. This WCEP, highlighted with solid arrows in Fig. 26.5a, has a WCET of 205 cycles. Assuming that some optimization is able to reduce `b`'s WCET from 80 down to 40 cycles, the CFG shown in Fig. 26.5b results from this optimization. As can be seen, the WCEP after optimization of `b` is `main`, `d`, and `c`. This example shows that the WCEP is very unstable during optimization – it can switch from one path within the CFG to a completely different one in the course of optimizations.

Thus, WCET-oriented and memory-aware compiler optimizations are faced with the challenges to always accurately model the current WCEP and to always be aware of possible WCEP switches. The following sections outline examples of WCET-oriented optimizations that exploit scratchpads and caches and that carefully consider WCEP switches. First, we present WCET-oriented SPM allocations to make the structural differences between memory-aware optimizations of energy dissipation (cf. Sect. 26.4) and of WCET estimates (cf. the following Sect. 26.5.1) evident. Next, we discuss cache locking optimizations in Sect. 26.5.2, followed by a presentation of cache partitioning for multitask systems in Sect. 26.5.3. Other approaches for timing models are explained in ▶ Chap. 19, "Host-Compiled Simulation" of this book. The importance of WCET-oriented optimizations for actual applications is stressed in Sect. 4 in ▶ Chap. 37, "Control/Architecture Codesign for Cyber-Physical Systems" of this book.

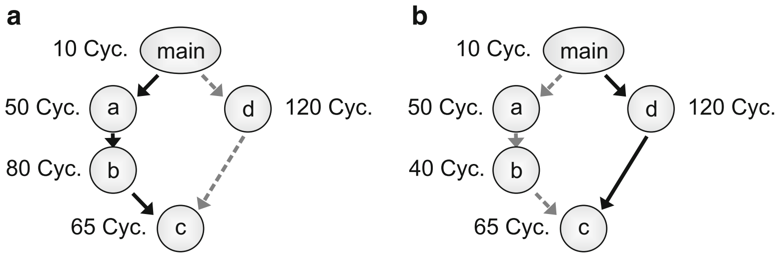

Fig. 26.5 (**a**) Original example CFG (**b**) Example CFG after optimization of `b`

26.5.1 WCET-Oriented Scratchpad Allocation

This section presents an ILP-based SPM allocation of program code that moves basic blocks statically onto the SPM [14, 39]. This is done under simultaneous consideration of possibly switching WCEPs by formulating ILP constraints that inherently model the longest path which starts at a certain basic block. The following equations use lowercase letters for ILP variables and uppercase letters for constants.

In analogy to the techniques presented previously in Sect. 26.4, the ILP also uses one binary decision variable v_i per basic block b_i of a program:

$$v_i = \begin{cases} 0 \text{ if basic block } b_i \text{ is assigned to } mem_{MAIN} \\ 1 \text{ if basic block } b_i \text{ is assigned to } mem_{SPM} \end{cases} \tag{26.10}$$

A scratchpad assignment is legal if the size of all basic blocks allocated to the SPM does not exceed the scratchpad's capacity. This property is ensured by adding in Equation (26.6) to the ILP again.

A block b_i of a function f causes some costs c_i, i.e., b_i's WCET$_{EST}$ depending on whether b_i is allocated to main memory or to the SPM:

$$c_i = C^i_{MAIN} * (1 - v_i) + C^i_{SPM} * v_i \tag{26.11}$$

Constants C^i_{MAIN} and C^i_{SPM} model the WCET$_{EST}$ values of b_i if it is executed from main memory or from the SPM, respectively. For reducible CFGs, an innermost loop l has exactly one back edge that turns it into a cyclic graph. Not considering this back edge turns l's CFG into an acyclic graph. This acyclic graph without l's back edge is denoted as $G_l = (V, E)$ here. Each node of G_l models a single basic block. Without loss of generality, there is exactly one unique exit node b^l_{exit} of loop l in G_l and one unique entry node b^l_{entry}. The WCET$_{EST}$ w^l_{exit} of b^l_{exit} is set to the costs of b^l_{exit}:

$$w^l_{exit} = c^l_{exit} \tag{26.12}$$

The WCET$_{EST}$ of a path from a node b_i (different from b^l_{exit}) to b^l_{exit} must be greater or equal than the WCET$_{EST}$ of any successor of b_i in G_l, plus b_i's costs:

$$\forall b_i \in V \setminus \{b^l_{exit}\} : \forall (b_i, b_{succ}) \in E : w_i \geq w_{succ} + c_i \tag{26.13}$$

Variable w^l_{entry} thus represents the WCET of all paths of loop l starting in b^l_{entry} if l is executed exactly once. To model several executions of l, all CFG nodes $v \in V$ of G_l are merged to a new super-node v_l. The costs of v_l are equal to l's WCET if executed once, multiplied by l's maximal loop iteration count C^l_{max}:

$$c_l = w^l_{entry} * C^l_{max} \tag{26.14}$$

Replacing a loop l by its super-node v_l may turn another loop l' of function f that immediately surrounds l into an innermost loop with acyclic CFG G'_l. Hence, Equations (26.12), (26.13), and (26.14) can be formulated analogously for l'. This way, the innermost loops of f are successively collapsed in the CFG so that ILP constraints that model f's control flow are created from the innermost to the outermost loops.

A program's WCEP can switch only at a block b_i with more than one successor because only there, forks in the control flow are possible. Since Equation (26.13) is created for each successor of b_i, variable w_i always reflects the WCET of any path starting from b_i – irrespective of which of the successors actually lies on the current WCEP. This way, Equation (26.13) realizes the implicit consideration of (switching) WCEPs in the ILP.

In analogy to the ILP modeling of loops, the WCET$_{EST}$ of a program's function f is represented by the variable w^f_{entry} if basic block b^f_{entry} is F's unique entry point. Whenever a basic block b_i calls some function f, variable w^f_{entry} is added to w_i in Equation (26.13) in order to model the interprocedural control flow correctly.

Finally, an entire C program's WCET$_{EST}$ is modeled by the ILP variable $w^{\texttt{main}}_{entry}$ that denotes the WCET$_{EST}$ of the program's unique entry point `main`. To minimize a program's WCET by the ILP, the following simple objective function is thus used:

$$\text{Minimize } w^{\texttt{main}}_{entry} \tag{26.15}$$

Furthermore, our ILP includes many additional constraints that take care of adjusted branch instructions making sure that a basic block located in main memory can still branch to a successor placed onto the SPM, and vice versa [14]. The discussion of these branching related constraints is omitted here for the sake of brevity.

This structure of the ILP can also be used to allocate global variables of a program onto the SPM. The main difference between the SPM allocation of code as described by Equations (26.10)–(26.15) and that of data objects is the cost modeling part. A binary variable x_j per data object d_j of a program specifies whether to allocate it to the SPM or not:

$$x_j = \begin{cases} 0 \text{ if data object } d_j \text{ is assigned to } mem_{MAIN} \\ 1 \text{ if data object } d_j \text{ is assigned to } mem_{SPM} \end{cases} \tag{26.16}$$

Here, the scratchpad capacity constraint (26.6) is simply formulated over the decision variables and sizes of the allocatable data objects. Again, each basic block b_i of a program causes some costs c_i. For the SPM allocation of data, these costs c_i reflect b_i's WCET$_{EST}$ depending on whether the data objects accessed by b_i are put in main memory or in the SPM:

$$c_i = C_i - \sum_{d_j \,\in\, \text{data objects}} G_{i,j} * x_j \tag{26.17}$$

Here, C_i denotes b_i's WCET$_{EST}$ if all data objects accessed by b_i are placed in main memory. $G_{i,j}$ is a constant that denotes the WCET reduction that b_i exhibits if data object d_j is put on the SPM. All other constraints (26.12), (26.13), (26.14), and (26.15) of the SPM allocation of program code that model the structure of a program's CFG remain unchanged when allocating data to the SPM.

Both ILP models are fully integrated into the WCET-aware C Compiler (WCC) [15, 47]. Due to the integration of a static WCET analyzer into the compiler, the constants C^i_{MAIN}, C^i_{SPM} and C_i used in Equations (26.11) and (26.17) are determined fully automatically. The sizes of basic blocks, data objects and SPM memories as well as the gain $G_{i,j}$ from Equation (26.17) are determined using WCC's processor-specific low-level intermediate representation. The maximal loop iteration counts C^l_{max} used in Equation (26.14) stem from WCC's polyhedral loop analyzer [26] or from user annotations. The WCC compiler infrastructure allows to generate the ILPs, their solution using IBM's `cplex` solver and the final memory allocation of the binary executable code in a fully automated fashion without any user intervention.

The following paragraphs show some experimental results that illustrate the WCET$_{EST}$ reductions that can be achieved by these two SPM allocations of program code and of data. Experiments have been performed for an Infineon TriCore TC1796 processor that features a 47 kB code SPM and a separate 40 kB data SPM that are both accessible within one clock cycle. The processor's main memory has an access latency of six clock cycles. WCET analyses were performed using the static timing analyzer aiT [1]. All results are generated using WCC's optimization level *-O2* so that our SPM allocations were applied to already highly optimized code.

We applied our SPM allocation of program code to 73 different real-life benchmarks. Code sizes range from 52 bytes up to 18 kB. Since these code sizes are much smaller than the totally available SPM size, we artificially limit the available SPM space for benchmarking. For each benchmark, SPM sizes of 10%, 20%, …, 100% of the benchmark's code size were used. Figure 26.6 shows the WCET estimates of all benchmarks produced by the WCET analyzer aiT that result from our SPM allocation as a percentage of the WCET$_{EST}$ when not using the program SPM at all. The bars in the diagram represent the average values over all 73 benchmarks. As can be seen, steadily decreasing WCET$_{EST}$ values were observed for increasing SPM sizes. Already for tiny SPMs with a capacity of 10% of a benchmark's code size, WCET$_{EST}$ decreases to 92.6% compared to the case

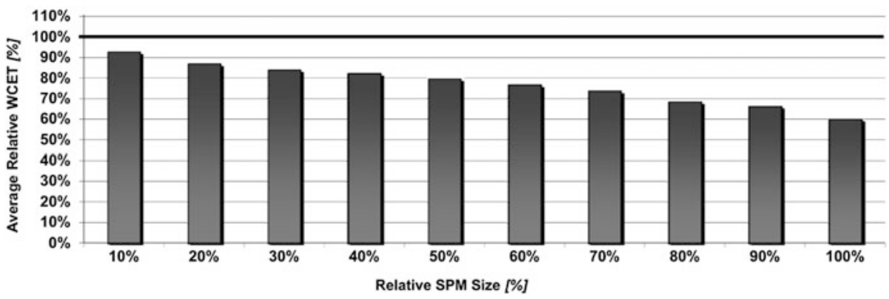

Fig. 26.6 Average relative WCET$_{EST}$ values after WCET-oriented SPM allocation of code

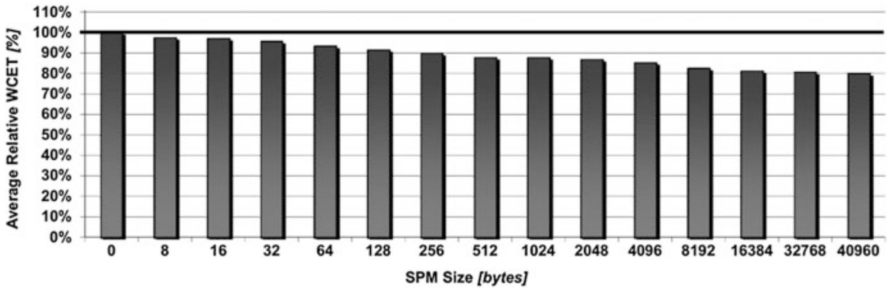

Fig. 26.7 Average relative WCET_{EST} values after WCET-oriented SPM allocation of data

when not using the SPM at all. For SPMs large enough to hold entire benchmarks, an average WCET_{EST} of only 60% of the original WCET was obtained. Thus, the achieved savings range between 7.4% and 40%. Our SPM allocation of program code potentially changes the benchmarks' code sizes due to the insertion of adjusted jump instructions in order to keep the control flow correct. It turned out that these changes are negligible – we observed a maximal code size increase by 128 bytes for our benchmarks. On average over all 73 benchmarks, code sizes increased by 0.02%.

Figure 26.7 shows the results of our SPM allocation of data averaged over all benchmarks that contain global data. The x-axis represents varying SPM sizes in absolute values. Again, we observed that WCET_{EST} decreased steadily for increasing data SPM sizes. Already for SPMs of only 8 bytes size, average WCET estimates over all benchmarks were reduced by 2.6%. For the real TriCore architecture with its 40 kB data SPM, average overall savings of 20.2% were achieved. The run-time complexity of both ILP-based SPM allocations is negligible in practice. ILP solving times of at most two CPU seconds were observed on an Intel Xeon at 2.4 GHz.

Both SPM allocations for code and data assume constant values to represent WCET values of basic blocks depending on the actual memory allocation (cf. Equations (26.11) and (26.17)). This in turn implies that the access latencies of the memories are also assumed to be constant like the six clock cycles for main memory accesses considered in this section. However, if Flash memory is used as main memory, its access latencies can vary, since Flash memory is organized in blocks and consecutive accesses within one block are faster than the six clock cycles used here. This behavior of Flash memories has no effect on the SPM allocation of code, since Equation (26.11) uses WCET values provided by a static timing analyzer that is inherently aware of the varying access latencies of Flash memories. The SPM allocation of data uses a constant gain $G_{i,j}$ for data memory accesses in Equation (26.17) which relies on the assumption of constant access latencies. Thus, this SPM allocation could take suboptimal allocation decisions so that its objective function from Equation (26.15) does not optimally minimize the global WCET of a program. However, since all WCET estimates used to generate Fig. 26.7 were solely obtained by aiT with its built-in support for Flash memories, our results can be considered safe, and the savings depicted in Fig. 26.7 are considerable despite of the conservative ILP model. We expect that the additional WCET reductions that could potentially be achieved by an improved ILP model considering block Flash accesses are marginal and not worth the effort.

26.5.2 Static Instruction Cache Locking

It is worthwhile mentioning that the structure of the ILP presented in the previous section is very general and flexible so that it can be employed to realize other memory-oriented optimizations beyond SPM allocation. The key difference between SPM allocation and cache locking is the granularity of the items to be allocated to the SPM or cache, respectively. In the case of SPMs, basic blocks or global variables of arbitrary size are candidates for memory allocation – cf. Equations (26.10) and (26.16). In contrast, the granularity of items that can be locked into a cache is defined by the cache's hardware architecture and its lockdown scheme.

For example, the ARM926EJ-S architecture supports way-based instruction cache locking which means that only complete columns of an N-way set-associative cache (cf. Sect. 26.3) can be locked. For an N-way set-associative cache with a total capacity of S_{CACHE} bytes, each way comprises S_{WAY} bytes:

$$S_{WAY} = S_{CACHE}/N \tag{26.18}$$

For a size B of a cache block given in bytes, the number of lines L per cache way is

$$L = S_{WAY}/B \tag{26.19}$$

Loading content from the main memory and locking it into a single cache line causes some architecture-specific but constant costs C_{LINE}. Thus, the costs for locking a complete way consisting of L lines are

$$C_{WAY} = L * C_{LINE} \tag{26.20}$$

Due to the modulo addressing of caches, memory addresses with $addr \bmod S_{WAY} \equiv 0$ are mapped to the beginning of a cache way. Thus, the main memory can be divided into memory blocks mb with a size of S_{WAY} bytes each such that each block can be entirely locked into a single cache way. This partitioning into blocks of size S_{WAY} is then applied to a program to be optimized by our cache locking approach – these blocks $mb_1, \ldots, mb_m$ denote candidates for cache locking. Thus, the ILP for instruction cache locking includes binary decision variables per memory block mb_j:

$$y_j = \begin{cases} 0 \text{ if memory block } mb_j \text{ remains unlocked} \\ 1 \text{ if memory block } mb_j \text{ is locked into the instruction cache} \end{cases} \tag{26.21}$$

An N-way set-associative cache can keep copies of up to N such memory blocks at the same time, since ways can only be locked in their entirety. Thus, an ILP constraint needs to ensure that the size of the contents locked into the cache does not exceed the cache size:

$$\sum_{j=1}^{m} y_j \leq N \tag{26.22}$$

As already shown in Sect. 26.5.1, each basic block b_i of a program causes some costs c_i. The WCET estimate of b_i if it is executed from main memory is denoted by the constant C^i_{MAIN} while C^i_{CACHE} represents its Worst-Case Execution Time if the block is locked into the cache. Given the size S_i of each basic block b_i and its start address in main memory, it is easy to determine the number of bytes $S_{i,j}$ of b_i that are part of memory block mb_j. Then, the potential WCET$_{EST}$ reduction of b_i in clock cycles $R_{i,j}$ if parts of it are executed from the cache due to a lockdown of mb_j is:

$$R_{i,j} = \frac{S_{i,j}}{S_i} * (C^i_{MAIN} - C^i_{CACHE}) \tag{26.23}$$

In the ILP for instruction cache locking, the costs c_i reflect b_i's WCET$_{EST}$ depending on whether memory objects that b_i is part of are locked into the cache:

$$c_i = C^i_{MAIN} - \sum_{j=1}^{m} y_j * R_{i,j} \tag{26.24}$$

Using these basic block costs c_i, the constraints from Equations (26.12), (26.13), (26.14) that model the structure of a program's CFG can, again, be reused without any further modification in order to realize the ILP for cache locking.

In analogy to Sect. 26.5.1, the WCET$_{EST}$ of a complete C program is represented by the ILP variable $w^{\texttt{main}}_{entry}$. However, static instruction cache locking as presented here involves some overhead in terms of WCET$_{EST}$, since some newly inserted code for loading and locking contents into the cache needs to be executed in the very beginning of function `main`. Thus, the objective function of the ILP for instruction cache locking that has to be minimized now models the WCET$_{EST}$ of the complete program including this lockdown overhead [34]:

$$\text{Minimize } w^{\texttt{main}}_{entry} + \sum_{j=1}^{m} y_j * C_{WAY} \tag{26.25}$$

This ILP model is again fully integrated and automated within the WCC compiler [15,47]. An evaluation has been carried out for an ARM926EJ-S processor that features a 16-kB large instruction cache with 32-byte line size, Least-Recently Used (LRU) replacement, and a configurable associativity of 2 or 4. Content can be accessed from the cache within one clock cycle, while main memory accesses

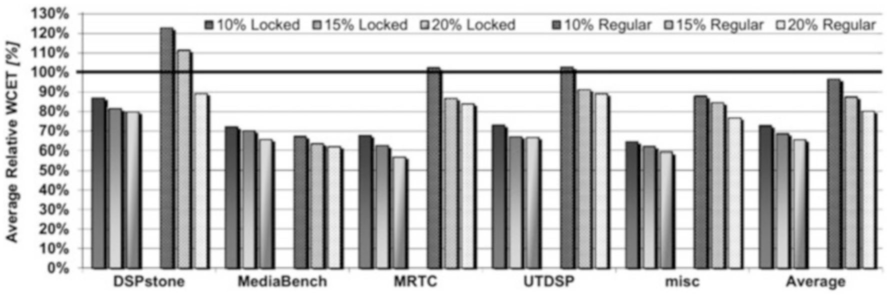

Fig. 26.8 Average relative WCET_{EST} values after WCET-oriented locking of 2-way set-associative instruction caches

take six cycles. The instruction cache supports way-based locking as described in this section. Loading and locking a single cache line of 32 bytes takes $C_{LINE} = 47$ clock cycles. The assumption of constant values for main memory access latencies and thus for the cache line locking overhead C_{LINE} has already been discussed at the end of Sect. 26.5.1. In the context of the instruction cache locking presented here, the imprecision of the ILP model is again considered marginal, since the constant locking overhead contributes exactly once to a benchmark's overall WCET, because the locking code is executed exactly once during the system startup phase.

Our cache locking optimization has been evaluated using 100 different real-life benchmarks from various commonly used benchmarking suites (DSPstone, MediaBench, MRTC, UTDSP, and some benchmarks from miscellaneous sources). For our evaluations, we artificially limited the cache sizes to 10%, 15%, and 20% of a benchmark's overall code size.

Figure 26.8 depicts the results of our static instruction cache locking scheme if applied to an architecture with a 2-way set-associative cache. The bars of the diagram show the WCET estimates that result from our cache locking as a percentage of the WCET_{EST} when executing the benchmarks without any cache. The figure shows average results over all used benchmark suites for the sake of readability. Per benchmark suite, detailed results are given for ILP-based cache locking as well as for a freely operating instruction cache without any locking. All locking-based results include the overhead for loading and locking blocks into the cache prior to a benchmark's execution. As can be seen from Fig. 26.8, ILP-based cache locking leads to maximal overall WCET_{EST} reductions of 35.4% (misc benchmarks) for very small caches with a capacity of 10% of a benchmark's code size. For caches of size 15% and 20%, respectively, the maximally achieved WCET_{EST} reductions increase up to 37.7% (misc benchmarks) and 43.1% (MRTC). The maximal improvements achieved by a regularly operating cache without locking amount to 32.6%, 36.3%, and 37.8% for the MediaBench suite and caches of sizes 10%, 15%, and 20%, respectively. Interestingly, our proposed cache locking always outperforms the regular caches except for MediaBench. Due to the static nature of the cache locking approach described here, the originally dynamic behavior of the cache gets lost. MediaBench exhibits a number of computation kernels that cannot be locked simultaneously into the size-restricted cache. In contrast, the regularly

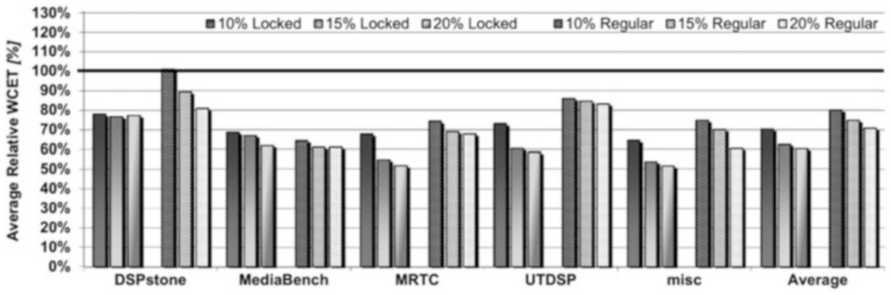

Fig. 26.9 Average relative WCET_{EST} values after WCET-oriented locking of 4-way set-associative instruction caches

operating cache can exchange the content during run time and adapt better to the characteristics of these benchmarks. Partial cache locking as described in [11] would be an alternative for such cases. For all other benchmark suites, the WCET_{EST} values resulting from our cache locking are significanly lower than those obtained by a regular cache. On average over all 100 considered benchmarks, locking leads to improvements of 27.1%, 31.2%, and 34.3% for 10%, 15%, and 20% large caches, respectively, while the unlocked caches of the same sizes only show improvements of 3.3%, 12.4%, and 19.6%, respectively.

The same trends were observed for a 4-way set-associative cache (cf. Fig. 26.9). Compared to the previous case with associativity of 2, the regular and unlocked 4-way set-associative cache achieves much better results due to its higher degree of freedom in which way to store some blocks: here, maximal WCET_{EST} reductions of 35.5%, 38.9% (MediaBench), and 39.5% (misc) were achieved. However, the locked cache still outperforms the unlocked one except for MediaBench. Cache locking leads to maximal improvements of 35.4%, 46.1%, and 48.3% (misc benchmarks) for caches of size 10%, 15%, and 20%, respectively. On average over all benchmarks, locking the 4-way set-associative cache improves WCET_{EST}s between 29.5% (10% cache size) and 39.6% (20% cache size) while the regular unlocked cache only achieves reductions from 19.8% (10% cache size) up to 29.2% (20% cache size).

Locking of content into data caches could be done in a similar fashion as described here. A first approach on static data cache locking using compile-time cache analysis was originally proposed in [42].

26.5.3 Instruction Cache Partitioning for Multitask Systems

While the techniques described so far are effective in reducing the WCET_{EST} of a single program, today's systems are often multitask systems where different programs are preempted and activated by a scheduler. For such multitask systems, caches are an even larger source of timing unpredictability as compared to single-task systems (cf. Sect. 26.3), because interrupt-driven schedulers lead to unknown points of time where task preemptions and context switches may happen.

Furthermore, it may happen that one task evicts cache contents belonging to some other task so that this other task exhibits additional cache misses if it resumes its execution. Finally, it is also unknown at which address the execution of a preempted task continues; hence it is unknown which cache set is accessed and eventually evicted next. Recent work on Cache-Related Preemption Delay (CRPD) analysis tries to incorporate scheduling and task preemption into timing analysis. But since the behavior of a cache in preemptive multitask systems cannot be predicted with 100% accuracy, the resulting WCET estimates are often highly overestimated, or some scheduling policies are not analyzable at all.

Cache partitioning is a technique to make the I-cache behavior perfectly predictable even for preemptive multitask systems. Here, the cache is divided into partitions of different sizes, and each task of a multitask system is assigned to one of these partitions. This partitioning is done such that each task can only evict entries from the cache that belong to its very own partition. By construction, a task can never evict cache contents of other tasks. As a consequence, multiple tasks do not interfere with each other any longer w.r.t. the cache during context switches. This allows to apply static WCET analyses for each individual task of the system in isolation. The overall WCET_{EST} of a multitask system using partitioned caches is then composed of the WCET_{EST} values of the single tasks given a certain partition size, plus the overhead required for scheduling and context switching.

Cache partitioning can be realized fully in software and thus does not require any support by the underlying cache hardware, as opposed to cache locking as presented in the previous Sect. 26.5.2. For this purpose, the code of each task has to be scattered over the main memory's address space in a way that it only uses such memory addresses that map to those cache sets that belong to the task's partition. Thus, a task's executable code is split into many chunks which are stored in non-consecutive regions in main memory. To make sure that a task's control flow remains correct after splitting it into chunks, additional jump instructions between these chunks need to be inserted. The generation of these chunks in the executable code can be done easily by the linker if a dedicated linker script describing this scattering and the different chunks is provided.

The remaining challenge consists of determining a partition size per task such that a multitask system's overall WCET_{EST} is minimized. In the following, preemptive round-robin scheduling of tasks is assumed and the period P_i of each task $t_i \in \{t_1, \ldots, t_m\}$ is known a priori. The length of the entire system's hyper-period is equal to the least common multiple of all tasks' periods P_i. The schedule count H_i then reflects the number of times that each task t_i is executed within a single hyper-period. Furthermore, a couple of n possible cache partition sizes $S_j \in \{S_1, \ldots, S_n\}$ measured in bytes is given beforehand.

WCET-aware software-based cache partitioning is modeled inside the WCC compiler using integer linear programming again [35]. A binary decision variable $z_{i,j}$ is used to model whether task t_i is assigned to a partition of size S_j:

$$z_{i,j} = \begin{cases} 0 \text{ if task } t_i \text{ is not assigned to a partition of size } S_j \\ 1 \text{ if task } t_i \text{ is assigned to a partition of size } S_j \end{cases} \tag{26.26}$$

The following constraints ensure that each task is assigned to exactly one partition:

$$\forall \text{ tasks } t_i \in \{t_1, \ldots, t_m\} : \sum_{j=1}^{n} z_{i,j} = 1 \tag{26.27}$$

In analogy to the SPM allocations presented in Sect. 26.5.1, the cache capacity constraint is given by:

$$\sum_{i=1}^{m} \sum_{j=1}^{n} z_{i,j} * S_j \leq S_{CACHE} \tag{26.28}$$

It is assumed here that the WCET$_{EST}$ $C_{i,j}$ of each task t_i if executed once using a cache partition of each possible size S_j is given a priori. This is achieved by performing a WCET analysis of each task for each partition size before generating the ILP for software-based cache partitioning. A task t_i's WCET$_{EST}$ c_i depending on the partition size assigned to the task by the ILP can thus be expressed as:

$$\forall \text{ tasks } t_i \in \{t_1, \ldots, t_m\} : c_i = \sum_{j=1}^{n} z_{i,j} * C_{i,j} \tag{26.29}$$

The objective function of the ILP models the WCET$_{EST}$ of the entire task set for one hyper-period. This overall WCET estimate to be minimized is thus defined by:

$$\text{Minimize} \sum_{i=1}^{m} H_i * c_i \tag{26.30}$$

Figure 26.10 shows the WCET estimates achieved by cache partitioning for three different benchmark suites. Since no multitask benchmark suites currently exist, randomly selected task sets from single-task benchmark suites were used.

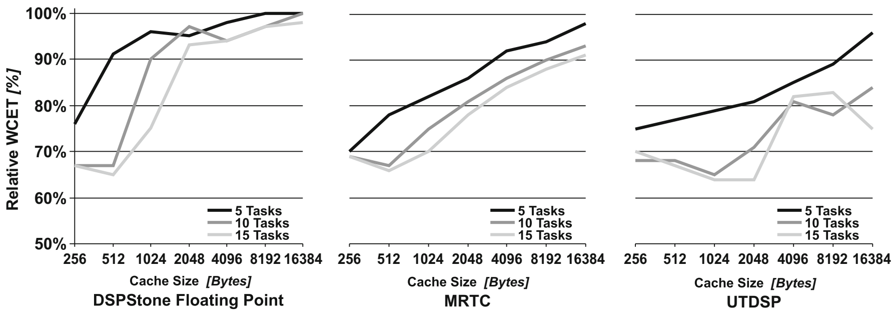

Fig. 26.10 Average relative WCET$_{EST}$ values after cache partitioning for multitask systems

Figure 26.10 shows results for task sets consisting of 5, 10, and 15 tasks, respectively. Each individual point in the figure's curves denotes the average value over 100 randomly selected task sets of a certain size. Benchmarking was done for an Infineon TriCore TC1796 processor with instruction cache sizes ranging from 256 bytes up to 16 kB. An access to the cache requires one clock cycle, accessing the main memory takes six cycles. All results are given as a percentage, with 100% corresponding to the $WCET_{EST}$ values achieved by a standard heuristic that uses a partition size per task which depends on the task's code size relative to the code size of the entire task set.

As can be seen, substantial $WCET_{EST}$ reductions of up to 36% were obtained. In general, WCET savings are higher for small caches and lower for larger caches. For DSPstone, $WCET_{EST}$ reductions between 4% and 33% were achieved. For the MRTC benchmarks, an almost linear correlation between $WCET_{EST}$ reductions and cache sizes was observed, with maximal WCET savings of 34%. For the large UTDSP benchmarks, $WCET_{EST}$ reductions of up to 36% were finally observed. In most cases, larger task sets exhibit a higher optimization potential so that cache partitioning achieves higher $WCET_{EST}$ improvements as compared to smaller task sets.

Software-based instruction cache partitioning as described in this section can be used for any processor and does not require any hardware support. However, hardware cache partitioning could be advantageous if dynamic repartitioning and adaptation of partition sizes at run time are desired. Such a dynamic partitioning scheme is difficult to realize in software, because it involves relocating the scattered code in memory at run time. As mentioned previously, additional jump instructions need to be added to the tasks' code in order to keep its control flow correct. The additional overhead contributed by these jumps is obviously the larger, the smaller the considered cache sizes are. For tiny caches of only 256 bytes, the overhead due to the additional jumps lies between 10% and 34% of the benchmark's total $WCET_{EST}$ for UTDSP and MRTC, respectively. For 16 kB large caches, the overhead lies between 1% and 2%.

A combination of partitioning and locking for data caches has been proposed in [43]. The authors use dynamic cache locking, static cache analyses and cache partitioning to ensure that all intratask conflicts, and consequently, memory access times, are exactly predictable.

26.6 Trade-Off Between Energy Consumption, Precision, and Run Time

26.6.1 Memory-Aware Mapping with Optimized Energy Consumption and Run Time

Thiele et al. designed the DOL tool for the optimized mapping of applications to multi-processor systems on a chip (MPSoCs) [40]. The original system is unaware of the sizes of the involved memory systems. Jovanovic modified this

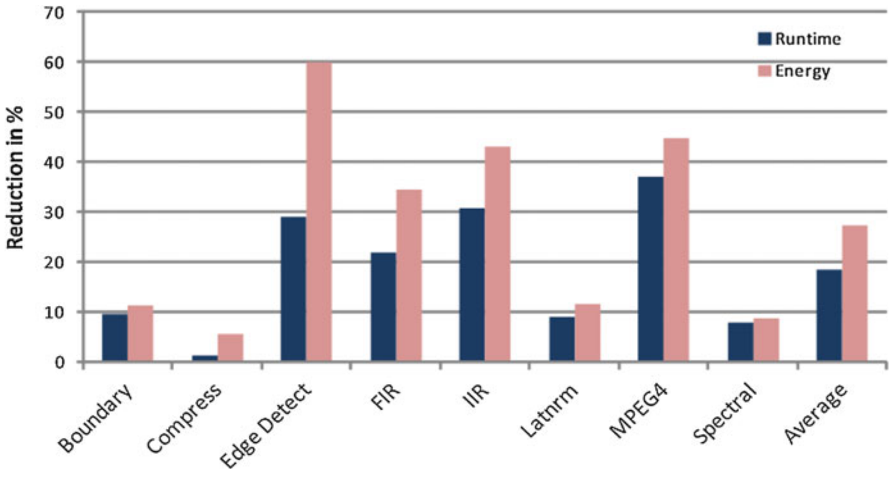

Fig. 26.11 Reduction in run time and energy achieved by run-time minimization

system such that the characteristics of available memories are also taken into account [20]. Also, the modified tool accepts general C-programs as input, instead of Kahn process networks. Input programs are generated by automatic parallelization [10]. Communication is based on First-In First-Out (FIFO) buffers. Two separate ILP models minimize either the energy consumption or the run time. The model minimizing the execution time includes access times of memories as well as expected execution times for the processors. Optimizations exploit available fast on-chip memories.

Figure 26.11 shows the reduction in run time and energy consumption for an ILP system minimizing run time. The baseline is an ILP-based mapping using run time as its objective.

In this case, the execution platform comprises four processors, each equipped with local data and instruction level-1 SPMs and a larger local level-2 memory. Furthermore, the platform includes a global shared memory which is used for communication. On average, memory awareness results in a reduction of the run time by 18% and of the energy by 27%.

Figure 26.12 shows the corresponding reduction for an ILP system minimizing the energy consumption. The baseline is an ILP-based mapping using energy as its objective. Compared to the results for run-time minimization, average run time is increased by 28%.

Both figures prove by means of an example that memory awareness allows a reduction of objectives run time and energy consumption. Also, minimization of run time does not automatically minimize energy consumption and vice versa, despite time being one variable in the computation of the energy consumption.

In a similar way, a trade-off between QoS and timeliness of results can be considered. For example, we can introduce qualifiers indicating whether or not

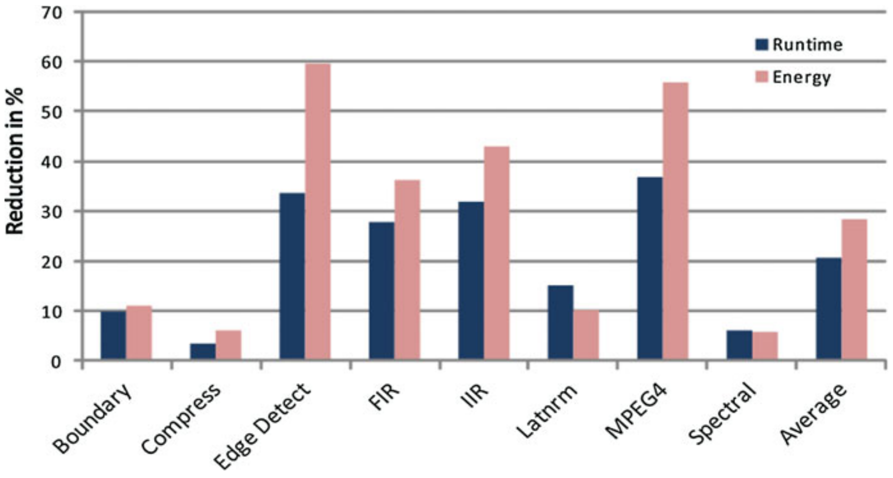

Fig. 26.12 Reduction in run time and energy achieved by energy minimization

variables should be allocated to reliable memory [37]. For variables not requiring reliable memory reads, we can skip error correction in the interest of timeliness of results.

26.6.2 Optimization for Three Objectives for the PAMONO Virus Sensor

In this section, we would like to demonstrate by means of an example how trade-offs between several objectives can be considered in such a way that reliable energy estimates are used. As an example, we will use a CPS for the detection of biological viruses based on Plasmon-Assisted Microscopy of Nano-Objects (PAMONO). The overall structure of the system can be seen in Fig. 26.13.

The sensor system includes an optical prism. On one of the sides (at the top in Fig. 26.13), there is a very thin gold layer covered with antibodies. Laser light entering through the second side of the prism, illuminating the backside of the gold layer and leaving through the third side is captured by a video camera. This prism is attached to a flow cell where the samples are applied. A pipe can be used to pump streams of gas or liquids across the gold layer. In case the stream contains viruses, they get stuck onto the gold layer with a certain probability. In case this happens, reflectivity of light reflected on the other side of the gold layer is affected and captured by the camera. Due to a resonance effect, the change is visible even when the size of the viruses is smaller than the wavelength of light. Real-time diagnosis of viruses like chicken-flu is among the potential applications.

However, due to the small dimensions of the camera sensor, video streams contain a significant amount of noise. A sophisticated image processing pipeline is needed in order to achieve a good detection quality. Figure 26.14 shows the pipeline

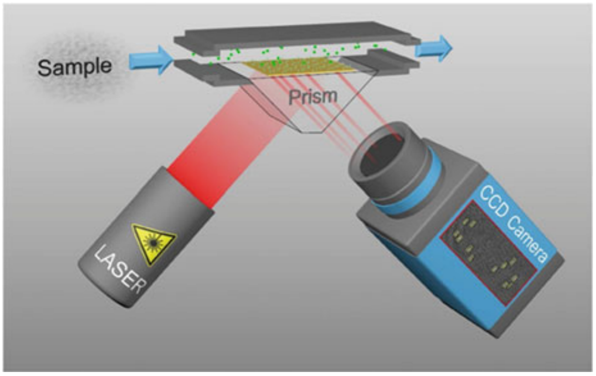

Fig. 26.13 Overall structure of the PAMONO virus sensor [33]

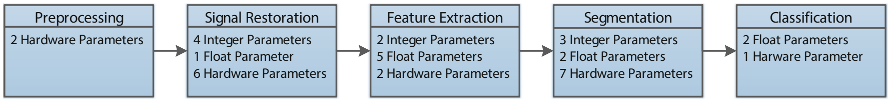

Fig. 26.14 Image processing pipeline to detect viruses. Listed parameters are modified by the GA to optimize the performance [33]

used in our research. In contrast to the previous section on WCET optimization, the application has soft real-time requirements.

In the pre-processing step, 16-bit gray-scale images are copied to the Graphics Processing Unit (GPU) and converted to floating-point arrays. In the next step, constant background noise is removed, and the signal of attaching virus is restored based on a sensor model of the PAMONO sensor. This step includes parameters which can be optimized for the best noise reduction under given circumstances. Various per-pixel and per-polygon features are computed during the feature extraction step. Per-pixel features describe the degree of membership to pixel classes representing virus adhesion. Per-polygon features perform the same function for polygons and their membership to polygon classes representing virus adhesion. During feature extraction, parameters comprise detection thresholds and parameters to switch between different feature extraction algorithms. Segmentation parameters control the way in which polygons are created and the way in which extracted features per pixel are combined to features per polygon. False classifications are minimized by appropriate classification parameters. The virus detection quality is measured with the F_1 score. This score is defined as the harmonic mean of the precision p and recall r:

$$F_1 = 2\frac{p \cdot r}{p + r} \tag{26.31}$$

with

$$\text{precision } p = \frac{TP}{TP + FP} \tag{26.32}$$

$$\text{and recall } r = \frac{TP}{TP + FN} \tag{26.33}$$

$$\begin{aligned} TP &: \text{ true positives} \\ FP &: \text{ false positives} \\ FN &: \text{ false negatives} \end{aligned}$$

One of the goals of our research was to demonstrate that the overall system can be downsized from a PC-based environment such that it can be operated even in environments with limited compute performance and power availability. For demonstration purposes, we selected the Odroid-XU3 [17] platform as an execution platform.

The XU3 contains two powerful multi-core processors with four cores each and a GPU resulting in a performance matching the needs of our application. The overall structure is shown in Fig. 26.15. Also and very importantly, it provides facilities for measuring the currents for all the cores and the memory. In this way, we can get around the problem of the limited precision of computer-based energy models. This allows considering energy during the optimization of the mapping of our application to the cores and the GPU. Unfortunately, the Odroid XU3 is superseded by the Odroid XU4 platform, which does not have this facility.

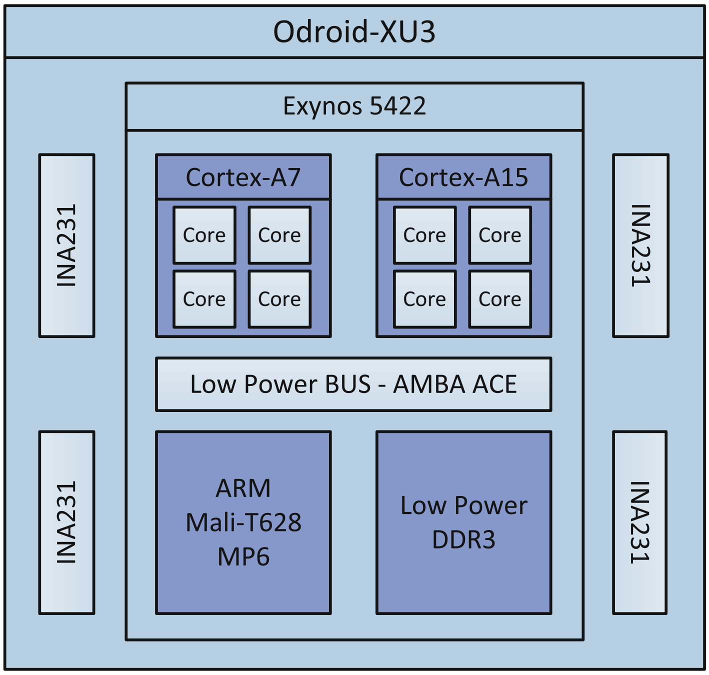

Fig. 26.15 Odroid execution platform [33]

A Genetic Algorithm (GA) is used to find Pareto-optimized design points considering execution times, energy consumption, and detection quality as objectives. (See ▶ Chap. 6, "Optimization Strategies in Design Space Exploration" of this book for a general discussion of Genetic Algorithm (GA)-based design space exploration. Other approaches for automatic parallelization and mapping to platforms can be found in ▶ Chaps. 28, "MAPS: A Software Development Environment for Embedded Multicore Applications" and ▶ 29, "HOPES: Programming Platform Approach for Embedded Systems Design" of this book.) The design space exploration is based on a heavily modified version of ECJ (Java-based Evolutionary Computation Research System) [28]. These modifications take parameter dependencies and parameter restrictions into account such that invalid parameter combinations are not generated. The evaluation of run times and energy consumption is based on the execution of the software on two available Odroid XU3 platforms concurrently as shown in Fig. 26.16. GPU; both Central Processing Units (CPUs) and memory energy consumptions are measured. Fitness results are averaged over several executions in order to remove jitter. Our energy measurement tool has been made publicly available [32].

An overview of the pipeline parameters is depicted in Fig. 26.14. Software parameters ranging from Boolean to restricted [0,1] floating-point values lead already to a large solution space. Beside pipeline parameters of the detection algorithm, several hardware parameters of our Odroid platform are considered by the optimization algorithm:

1. The used governor controlling operating parameters at run time (e.g., performance, powersave, interactive)
2. The frequency (200 MHz to 2 GHz in 100 MHz steps) of the Cortex-A15 core, if control is allowed by the governor
3. Work group sizes of all pipeline elements mapped to the Mali-T628 GPU
4. The memory allocation size for buffers storing among other things the detected polygons on the GPU

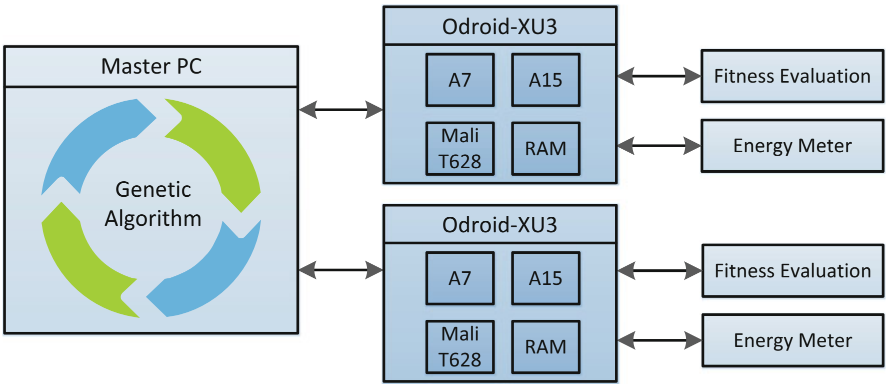

Fig. 26.16 Evolutionary optimization process [33]

In the following, we will focus on parameters dealing with memory configurations. The work group sizes on the Mali-T628 GPU affect the number of threads concurrently running on the GPU and thus have major impact on how fast the data can be processed and how much energy is consumed by the GPU. Partial results within the work group are shared by synchronizing using the shared memory on the streaming multi-processor on the GPU. Thus, memory restricts the parallelism which could be extracted. In addition, the memory allocation size for some of the buffers on the GPU can affect the detection quality. Within the different pipeline steps, ring buffers on the GPU store some of the previously processed images. Depending on the ring buffer sizes, the number of available images varies, e.g., for noise reduction or the feature extraction, which increases/decreases the quality of the results.

To give a detailed example on memory (buffer) allocation optimization, we will now focus on the application of the sensor model and temporal noise reduction applied to the captured images to identify possible virus pixels. According to the PAMONO sensor model, a captured image consists of a background signal, multiplied with a virus signal and an additive noise term. A potential virus pixel can be identified by an increase in intensity. Thus, a sliding window of size b of the past and a sliding window of size a of the future are used to detect potential virus pixels. Since the virus-binding process takes some time, it is not to be seen instantaneously. Thus, a time interval of size g is used to model this attaching process. Figure 26.17 shows the intensity of one pixel over a time series of frames. The detection algorithm calculates the median (red horizontal line) over b and a images. If the difference between the two medians exceeds a specific threshold, this pixel is considered

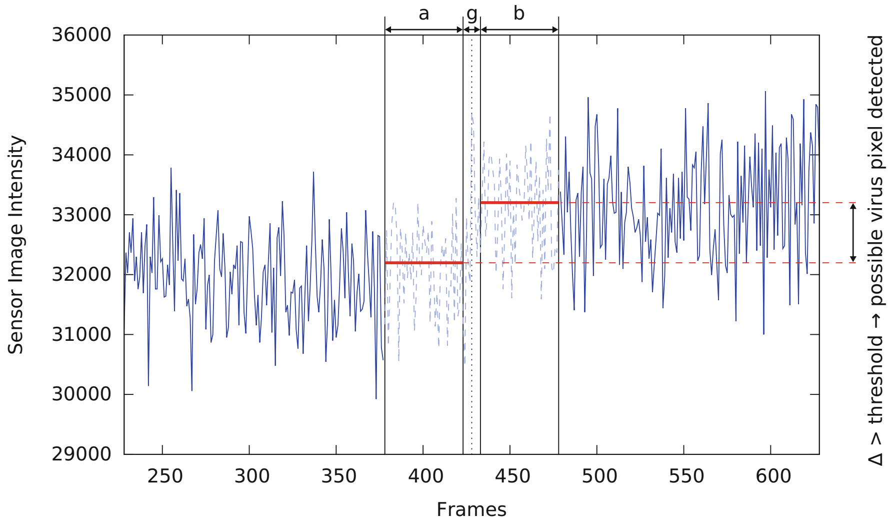

Fig. 26.17 Sensor model application and temporal noise reduction for one pixel and one time step to detect possible virus pixels

as a possible virus pixel. As time moves forward, this is comparable to a sliding window moving over large data. For this preprocessing step, $a + b + g$ images need to be stored in GPU's memory and intuitively, increasing the intervals increases the performance of this processing step. However, the memory is limited, and determining a good combination regarding the other memory parameters in the pipeline is complex. Thus, the genetic algorithm takes care of this selection process. In later pipeline stages, all possible virus pixels are analyzed in more detail, e.g., additional per-pixels and polygon features are extracted.

For the evaluation, we used two data sets for the virus detection program. We used a training and a testing data set, each consisting of 1,000 16-bit gray-scale sensor images with size 706 pixels × 167 pixels. Both data sets were labeled, thus the correct positions of all virus pixels are known, and we can calculate the F_1 score according to Equation (26.31). We conducted three different experiments. Firstly, only hardware parameters were optimized. Secondly, only software parameters were optimized. Thirdly, hardware and software parameters were optimized simultaneously. The unoptimized detection software achieves 7.5 frames per second while reaching the best detection quality. The greatest improvements could be observed for the combined optimization. Figure 26.18 shows the result of this combined optimization experiment. For execution time and energy consumption,

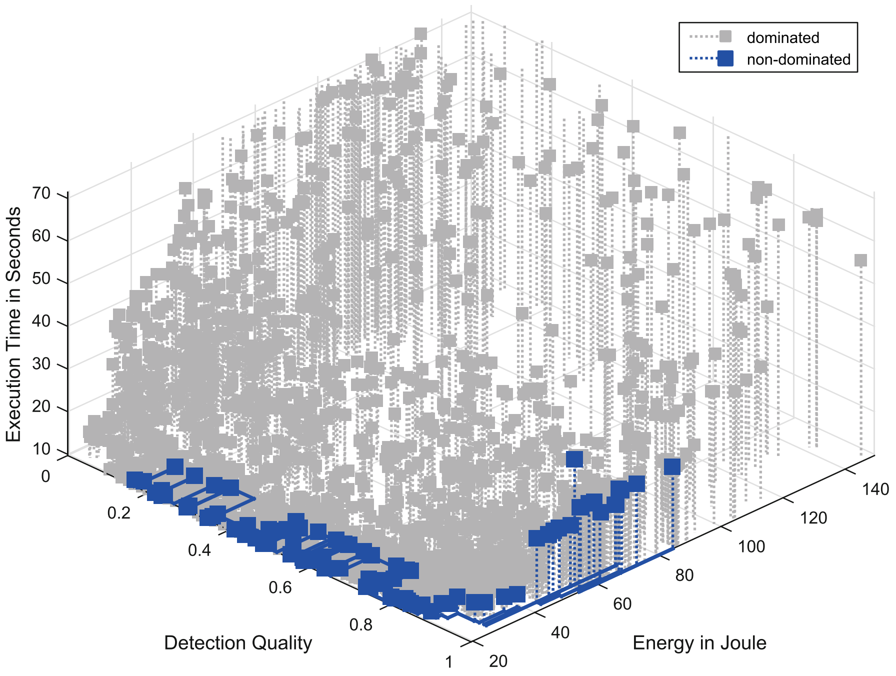

Fig. 26.18 Trade-off between detection quality, energy consumption, and run time resulting from an optimization of hardware and image pipeline parameters. For execution time and energy consumption, lower values are better, and for detection quality, higher values are better [33]

Table 26.1 Excerpt of the Pareto front for the objectives virus detection quality (F_1 training), energy consumption, and execution time. In addition, the detection quality (F_1 testing) for the unseen testing data set is shown. As baseline/comparative measurement, an unoptimized run is given in the first row, which was measured with an unmodified system and program [33]

F_1 training	F_1 test	Energy cons.	Energy sav.	Exec. time	Speedup	Frame rate
100% (fixed)	99.5% (fixed)	370.0 Joule	–	119.8 s	–	7.5 fps
100%	99.5%	57.5 Joule	84%	29.3 s	4.1	30.7 fps
100%	99.5%	84.5 Joule	77%	28.9 s	4.1	31.1 fps
98.5%	97.4%	47.9 Joule	87%	25.5 s	4.7	35.3 fps
97.4%	99.5%	69.3 Joule	81%	23.9 s	5.0	37.7 fps
96.9%	87.8%	27.7 Joule	93%	14.8 s	8.1	60.8 fps
87.9%	76.6%	22.3 Joule	94%	10.8 s	11.1	83.3 fps
84.2%	60.5%	20.7 Joule	94%	11.4 s	10.5	78.9 fps
74.2%	63.9%	23.5 Joule	94%	10.7 s	11.2	84.1 fps
74.2%	64.7%	33.6 Joule	91%	10.4 s	11.5	86.5 fps
51.9%	55.8%	33.0 Joule	91%	10.0 s	12.0	90.0 fps

lower values are better, and for detection quality, higher values are better. The Pareto front is highlighted, and Table 26.1 shows an excerpt of it. Without losing quality, for example, a solution running at 30.7 fps with 84% energy savings was generated. A not 100% detection quality might be sufficient to prove that a sample is contaminated with viruses. By accepting loss of quality, even higher frame rates and energy savings were observed. For example, a solution which still achieves a good detection quality like 76.6% results in an energy saving of 94% and a speedup of more than 11. This indicates that one could use a less capable and thus cheaper hardware or increase the resolution of the camera sensor. An increased resolution enables the simultaneous detection of different virus types. Here, the gold layer is partitioned with different antibodies, and an increased resolution is necessary to detect viruses.

Flexibility with respect to the detection quality is the new objective in this example. This example demonstrates that, in the future, we should not just optimize the usage of the memory in isolation. Rather, it should be included in an overall optimization process for several objectives. This optimization needs to include both the software compilation process as well as the exploration of hardware parameters. This example also demonstrates that hardware parameters exist even in the case of off-the-shelf hardware.

26.7 Conclusions and Future Work

In this chapter, we have demonstrated consequences of the fact that the size of memories has a large impact on their access times and energy consumption. This impact leads to heterogeneous memory architectures comprising a mixture of fast small memories and relatively slow larger memories. Other consequences are resulting from the fact that information processing in Cyber-Physical Systems has to take a number of objectives and constraints into account. This leads to the idea of an exploitation of memory characteristics such that constraints are met and objectives are used for optimizations. In this chapter, we have presented results of our research groups. First results concern the optimized use of Scratchpad memories for a reduction of the energy consumption. Detailed results are presented for the case of hard real-time systems: we present compiler optimizations using the Worst-Case Execution Times as their objective. We are also briefly describing the optimized mapping to multi-core platforms with a choice of objectives to optimize for. Finally, we demonstrate the integration of memory optimizations into a global approach for the optimization of a cyber-physical sensor system with soft deadlines. In this case, the scope for optimizations comprises software and hardware parameters. The goal is to find good trade-offs between multiple objectives, including quality of service.

We believe that this chapter demonstrates trends in the design of embedded and CPS very nicely. There is a trend from the consideration of just the memory system for mono-processors and single objectives towards whole system optimization for multi-core systems for multiple objectives. The inclusion of the Quality of Service as an objective offers new opportunities, since we can trade off the quality of service against other objectives.

Acknowledgments Part of the work on this section has been supported by Deutsche Forschungsgemeinschaft (DFG) within the Collaborative Research Center SFB 876 "Providing Information by Resource-Constrained Analysis," project B2. URL: http://sfb876.tu-dortmund.de

References

1. AbsInt Angewandte Informatik GmbH (2016) aiT: Worst-Case Execution Time Analyzers. http://www.absint.com/ait
2. AbsInt Angewandte Informatik GmbH (2016) Stack overflow is a thing of the past. https://www.absint.com/stackanalyzer/index.htm
3. Amdahl GM (1967) Validity of the single processor approach to achieving large scale computing capabilities. AFIPS spring joint computer conference
4. Bai K, Shrivastava A (2010) Heap data management for limited local memory (LLM) multi-core processors. In: Proceedings of the international conference on hardware/software codesign and system synthesis (CODES+ISSS), pp 317–325
5. Banakar R, Steinke S, Lee BS, Balakrishnan M, Marwedel P (2002) Scratchpad memory: a design alternative for cache on-chip memory in embedded systems. In: Proceedings of the international symposium on hardware-software codesign (CODES), Estes Park (Colorado)
6. Borkar S (2005) Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. IEEE Micro 25(6):10–16
7. Burks A, Goldstine H, von Neumann J (1946) Preliminary discussion of the logical design of an electronic computing element. Report to U.S. Army Ordnance Department, reprinted at https://www.cs.princeton.edu/courses/archive/fall10/cos375/Burks.pdf
8. Chang DW, Lin IC, Chien YS, Lin CL, Su AWY, Young CP (2014) CASA: contention-aware scratchpad memory allocation for online hybrid on-chip memory management. IEEE Trans Comput-Aided Des Integr Circuits Syst 33(12):1806–1817. doi:10.1109/TCAD.2014.2363385
9. Cho H, Egger B, Lee J, Shin H (2007) Dynamic data scratchpad memory management for a memory subsystem with an MMU. In: Proceedings of the conference on languages, compilers, and tools for embedded systems (LCTES). ACM, New York, pp 195–206. doi:10.1145/1254766.1254804

10. Cordes D, Engel M, Neugebauer O, Marwedel P (2013) Automatic extraction of pipeline parallelism for embedded heterogeneous multi-core platforms. In: Proceedings of the international conference on compilers, architectures, and synthesis for embedded systems (CASES), Montreal
11. Ding H, Liang Y, Mitra T (2012) WCET-centric partial instruction cache locking. In: Proceedings of the design automation conference (DAC), San Francisco
12. Dominguez A, Udayakumaran S, Barua R (2005) Heap data allocation to scratch-pad memory in embedded systems. J Embed Comput 1(4):521–540
13. Drepper U (2007) What every programmer should know about memory. http://www.akkadia.org/drepper/cpumemory.pdf
14. Falk H, Kleinsorge JC (2009) Optimal static WCET-aware scratchpad allocation of program code. In: Proceedings of the design automation conference (DAC), San Francisco, pp 732–737
15. Falk H, Lokuciejewski P (2010) A compiler framework for the reduction of worst-case execution times. Int J Time Crit Comput Syst (Real Time Syst) 46(2):251–300
16. Falk H, Verma M (2004) Combined data partitioning and loop nest splitting for energy consumption minimization. In: Proceedings of the international workshop on software and compilers for embedded systems (SCOPES), Amsterdam, pp 137–151
17. Hardkernel Co., Ltd., Odroid-XU3. http://www.hardkernel.com/main/products/prdt_info.php?g_code=G140448267127 (2015)
18. Hofmann M, Jost S (2003) Static prediction of heap space usage for first-order functional programs. In: Proceedings of the symposium on principles of programming languages (POPL). ACM, New York, pp 185–197. doi:10.1145/604131.604148
19. HP Labs, CACTI – an integrated cache and memory access time, cycle time, area, leakage, and dynamic power model. http://www.hpl.hp.com/research/cacti/ (2015)
20. Jovanovic O, Kneuper N, Marwedel P, Engel M (2012) ILP-based memory-aware mapping optimization for MPSoCs. In: Proceedings of the conference on embedded and ubiquitous computing (EUC), Paphos, Cyprus
21. Kang S, Dean AG (2012) Leveraging both data cache and scratchpad memory through synergetic data allocation. In: Proceedings of the real time and embedded technology and applications symposium (RTAS). IEEE Computer Society, Washington, DC, pp 119–128. doi:10.1109/RTAS.2012.22
22. Kannan A, Shrivastava A, Pabalkar A, Lee JE (2009) A software solution for dynamic stack management on scratch pad memory. In: Proceedings of the Asia and South Pacific design automation conference (ASPDAC), pp 612–617
23. Kotthaus H, Korb I, Marwedel P (2015) Performance analysis for parallel R programs: towards efficient resource utilization. Technical Report 1/2015, TU Dortmund, CS Department
24. Li L, Wu H, Feng H, Xue J (2007) Towards data tiling for whole programs in scratchpad memory allocation. In: Proceedings of the Asia-Pacific conference on advances in computer systems architecture (ACSAC). Springer, Berlin/Heidelberg, pp 63–74. doi:10.1007/978-3-540-74309-5_8
25. Liu Y, Zhang W (2015) Scratchpad memory architectures and allocation algorithms for hard real-time multicore processors. J Comput Sci Eng 9:51–72
26. Lokuciejewski P, Cordes D, Falk H, Marwedel P (2009) A fast and precise static loop analysis based on abstract interpretation, program slicing and polytope models. In: Proceedings of the international symposium on code generation and optimization (CGO), Seattle, pp 136–146
27. Luican II, Zhu H, Balasa F (2006) Formal model of data reuse analysis for hierarchical memory organizations. In: Proceedings of the international conference on computer-aided design (ICCAD). ACM, New York, pp 595–600. doi:10.1145/1233501.1233623
28. Luke S, Panait L, Balan G, Paus S, Skolicki Z, Popovici E, Sullivan K, Harrison J, Bassett J, Hubley R (2015) ECJ: a java-based evolutionary computation research system. http://cs.gmu.edu/~eclab/projects/ecj/
29. Marwedel P (2010) Embedded system design – embedded systems foundations of cyber-physical systems. Springer, New York

30. McIlroy R, Dickman P, Sventek J (2008) Efficient dynamic heap allocation of scratch-pad memory. In: Proceedings of the international symposium on memory management, pp 31–40
31. National Science Foundation (2013) Cyber-physical systems (CPS). http://www.nsf.gov/pubs/2013/nsf13502/nsf13502.htm
32. Neugebauer O, Libuschewski P (2015) Odroid energy measurement software. http://sfb876.tu-dortmund.de/auto?self=Software
33. Neugebauer O, Libuschewski P, Engel M, Mueller H, Marwedel P (2015) Plasmon-based virus detection on heterogeneous embedded systems. In: Proceedings of the international workshop on software and compilers for embedded systems (SCOPES)
34. Plazar S, Falk H, Kleinsorge JC, Marwedel P (2012) WCET-aware static locking of instruction caches. In: Proceedings of the international symposium on code generation and optimization (CGO), San Jose, pp 44–52
35. Plazar S, Lokuciejewski P, Marwedel P (2009) WCET-aware software based cache partitioning for multi-task real-time systems. In: Proceedings of the international workshop on worst-case execution time analysis (WCET), Dublin, pp 78–88
36. Pyka R, Fassbach C, Verma M, Falk H, Marwedel P (2007) Operating system integrated energy aware scratchpad allocation strategies for multi-process applications. In: Proceedings of the international workshop on software and compilers for embedded systems (SCOPES)
37. Schmoll F, Heinig A, Marwedel P, Engel M (2013) Improving the fault resilience of an H.264 decoder using static analysis methods. ACM Trans Embed Comput Syst (TECS) 13(1s):31:1–31:27. doi:10.1145/2536747.2536753
38. Steinke S, Wehmeyer L, Lee BS, Marwedel P (2002) Assigning program and data objects to scratchpad for energy reduction. In: Proceedings of design, automation and test in Europe (DATE)
39. Suhendra V, Mitra T, Roychoudhury A, et al. (2005) WCET centric data allocation to scratchpad memory. In: Proceedings of the real-time systems symposium (RTSS), Miami, pp 223–232
40. Thiele L, Bacivarov I, Haid W, Huang K (2007) Mapping applications to tiled multiprocessor embedded systems. In: International conference on application of concurrency to system design, pp 29–40. doi:10.1109/ACSD.2007.53
41. Udayakumararan S, Dominguez A, Barua R (2006) Dynamic allocation for scratch-pad memory using compile-time decisions. ACM Trans Embed Comput Syst (TECS) 5:472–511
42. Vera X, Lisper B, Xue J (2003) Data cache locking for higher program predictability. ACM SIGMETRICS Perform Eval Rev 31(1):272–282
43. Vera X, Lisper B, Xue J (2007) Data cache locking for tight timing calculations. ACM Trans Embed Comput Syst (TECS) 7(1):1–38
44. Verma M, Marwedel P (2006) Overlay techniques for scratchpad memories in low power embedded processors. IEEE Trans Very Large Scale Integr Syst 14(8):802–815
45. Wang P, Sun G, Wang T, Xie Y, Cong J (2013) Designing scratchpad memory architecture with emerging STT-RAM memory technologies. In: Proceedings of the international symposium on circuits and systems (ISCAS), pp 1244–1247. doi:10.1109/ISCAS.2013.6572078
46. Wang Z, Gu Z, Yao M, Shao Z (2015) Endurance-aware allocation of data variables on NVM-based scratchpad memory in real-time embedded systems. IEEE Trans Comput-Aided Des Integr Circuits Syst 34(10):1600–1612. doi:10.1109/TCAD.2015.2422846
47. WCET-aware Compilation (2016) http://www.tuhh.de/es/esd/research/wcc
48. Zhang W, Ding Y (2013) Hybrid SPM-cache architectures to achieve high time predictability and performance. In: Proceedings of the conference on application-specific systems, architectures and processors (ASAP), pp 297–304. doi:10.1109/ASAP.2013.6567593