An Introduction to the Research on Scratchpad Memory: Definition, Hardware, Known Implementations and WCET Optimisation

Julius Roob
University of Kaiserslautern, Embedded Systems Group
julius@juliusroob.de

Contents

1 Introduction 2
1.1 Definition: Scratchpad Memory .................................................. 2
1.2 Contents of this paper ............................................................... 3
1.3 Motivation ................................................................. 3

2 Energy Efficiency 3

3 Known applications/implementations of scratchpad memory Systems 4
3.1 Microcontrollers ............................................................... 4
3.2 Cuda and OpenCL .............................................................. 4
3.3 Emotion Engine[10] ............................................................. 5
3.4 Cell Architecture ................................................................. 5
    3.4.1 Code Overlay .............................................................. 6

4 Cache Locking 7
4.1 Architectures ................................................................. 7
4.2 Cache Locking vs. Scratchpad ............................................... 7

5 Scratchpad in real-time systems 8
5.1 Caches ................................................................. 9
5.2 WCET centric data allocation to scratchpad memory[15] .......... 9
    5.2.1 ILP ................................................................. 9
    5.2.2 Knapsack? ............................................................ 10
    5.2.3 Branch and Bound ..................................................... 10
    5.2.4 Greedy Heuristic ..................................................... 11
    5.3.1 ILP ................................................................. 12
    5.3.2 Compiler ............................................................. 13
    5.3.3 Evaluation ............................................................. 14
    5.4.1 Definition of problem .................................................. 14
    5.4.2 Choice of reload points .............................................. 15
Abstract

Together with [13], this paper aims to be an introduction to the concept and a summary of the current research on the topic of Scratchpad Memory (SPM). The topics I focus on are the hardware aspects, energy efficiency, both the general concept of SPM and known implementations and applications, as well as the issue of worst-case execution time in hard real-time systems.

1 Introduction

1.1 Definition: Scratchpad Memory

Scratchpad Memory (SPM) is the term chosen for cache-like software-managed memory. It is significantly smaller than the main memory, ranging from below 1-KB to several KB in research applications and being at 256-KB in the SPEs of the Cell (section 3.4) multiprocessor. Being located on the same chip as - and close to - the CPU core, its access latencies are negligible compared to those of the main memory.

Unlike caches, SPM is not transparent to software. It is mapped into an address range different from the external RAM, which is outlined in figure [1]

![Figure 1: Cache and SPM memory model](image)

Some implementations make it possible for the CPU to continue its calculations while data is transferred from RAM to SPM or vice versa by employing an asynchronous DMA controller. Even without it being asynchronous, transfers from or to RAM are often handled by a special controller that moves data in blocks rather than having the CPU using load and store instructions.

There are approaches that use both a SPM and a regular cache.
In multicore processors, there may be a separate SPM per core, which can, depending on the implementation, be used as private buffer memory, ease communication between cores or both. (see section 3.4 for an example).

1.2 Contents of this paper

After the definition and motivation of SPM, in section 2 I will discuss the hardware details and impact on energy efficiency. Section 3 will name examples for known applications, with a focus on the Cell multiprocessor. A short explanation of cache locking, as well as some implemented examples are given in section 4 together with a comparison of the concepts. Finally, section 5 gives an introduction to SPM in WCET optimisations, summarising three papers on both static allocation of data and code, as well as dynamic cache locking.

1.3 Motivation

Modern computer applications require more RAM to perform tasks than can be embedded into the processor core. Apart from some low-power embedded systems, most processors utilise cache hierarchies to lessen the speed penalty caused by access to external memory. Cache is a small temporary buffer managed by hardware, employing usually hard-wired displacement strategies like least-recently-used (LRU), first-in-first-out (FIFO) or randomised approaches.

Since these displacement strategies are written to perform good for a wide spectrum of use cases, they are less optimal than a strategy that is tailored to a specific application by a compiler that knows about the whole program structure and may even employ profiling data.

Furthermore, because of its lack of most of the management logic cache requires, SPM is less demanding both in chip area and complexity. This will be further explained in section 2.

WCET calculation in hard real-time systems is easier and provides tighter estimates when employing SPM, since it is more predictable and gives developers or compilers more possibilities to optimise.

2 Energy Efficiency

The main advantage of cache over SPM is that it is transparent to software. To achieve this, it needs to know which memory addresses lie within blocks that are currently stored in the cache. Tags are the parts of memory addresses that are required to map a block of cache memory to the address in the RAM it belongs to. They are stored next to the cache blocks they belong too in so-called tag lines, see figure 2. Next to the tag lines and the necessary logic to determine whether a memory access is already cached, a controller that fetches previously uncached memory blocks and takes care of block displacement is required. Depending on the implementation, there may be different mechanisms like write-back and write-through as well as several displacement strategies available that applications or the operating system can choose from.

Since the on-chip cache is a major part in the energy consumption of a modern processor, requiring from 25% to 45%, increasing its efficiency or replacing it with SPM has a significant impact on the energy consumption of the whole processor.

To compare the energy and area efficiency of SPM and cache, [16] modifies an existing processor, the ARM7TDMI, to use an SPM instead of the previously built-in cache. They employ the energy-away encc compiler with the post pass option of assigning code and data blocks with the knapsack algorithm. After optimised compilation, the resulting executable is emulated using ARMulator which emits a trace
of all memory access operations. Those can be used to determine both the energy consumption of the SPM and the cache.

The observed average reduction in time and area are 34% and 18% for constant cycle times and SPM is about 3 times more efficient than a 4-way set associative cache when energy is concerned.

Utilising profiling and graph partitioning to optimise SPM allocation as well as a custom management unit with an instruction to load code into the SPM, [6] achieved a 50.7% energy reduction and an 53.2% improve in performance.

3 Known applications/implementations of scratchpad memory Systems

This section is dedicated to widespread architectures whose memory model fits the definition of scratchpad memory.

3.1 Microcontrollers

Both the Atmel megaAVR[1] and the STMicroelectronics STM32 ARM Cortex based microcontroller have interfaces to connect external memory. These extend the internal ram and are mapped at own address ranges in the address space available to the processor.

Because the on-chip memory for those is an order of magnitude faster and smaller than the external RAM, it can be considered a scratchpad memory.

3.2 Cuda and OpenCL

Cuda is a framework for general purpose GPU programming (GPGPU) developed by NVIDIA and is restricted to their GPU architectures. OpenCL is an open standard designed to be portable to more architectures, including even multicore CPUs and Cell (section 3.4). Both can benefit from Scratchpad Memory Optimisation as well.

The GPU architectures are outlined in [5] and [4].

Software written for both Cuda or OpenCL has to explicitly manage the allocation of contents and transfers of those between different memory levels. Before being used for calculations, data has to be transferred between the RAM of the host system and the memory built onto the graphics adaptor or
other accelerator device. A main problem of GPGPU calculations is the latency and bandwidth of those transfers, since, especially with GPUs, they are an order of magnitude slower than on-device operations.

3.3 Emotion Engine

The Emotion Engine powering the PlayStation 2 featured a 16-KB scratchpad memory, used mainly to improve communication between the CPU and both the floating-point SIMD-VLIW processors.

3.4 Cell Architecture

The Cell multiprocessor employed in the PlayStation 3 as well as high-performance clusters features two approaches for software-managed memory hierarchy.

It was developed by Sony, IBM and Toshiba with the purpose of establishing a platform that provides high performance, energy efficiency and support for real-time applications.

One Cell processor consists of one PowerPC Element and 8 Synergistic Processing Elements(SPE), the structure of which is outlined in figure 5.

A single SPE consists of 256-KB local storage, a SIMD Synergistic Processing Unit(SPU) and a DMA controller. The local storage fits the definition of Scratchpad Memory since both the content of the local storage and the DMA transfers from and to RAM are software-controlled. Those DMA transfers are asynchronous, which means there can be up to 16 queued DMA transfers without forcing the SPU to wait for their completion. Being unable to access memory outside of their local storage without DMA transfers, the code running on the SPU may be required to be split into overlays, the details of which will be discussed in section 3.4.1.
The local storage of each SPE is mapped into the global address space to allow both the other SPEs and the PPE to transfer data from and to it. This allows programmers to choose the data flow model that best fits the application, for example they can organise their SPEs into a chain, in which data is efficiently streamed from one element to the next, being processed at every station. The PowerPC Element embedded in the Cell multiprocessor is a general purpose CPU that allows a Operating System to manage the other processors. It posses 32-KB first-level instruction and data caches as well as a 512-KB second-level cache, the latter of which employs replacement management tables that allow the operating system to lock contents to cache.

### 3.4.1 Code Overlay

Without transferring data through the DMA controller of its SPE, each SPU can only access and execute the contents of the local storage associated with it. [7]

Code Overlay [3] is a mapping technique to execute programs that are larger than the available memory (local storage) by splitting that memory into regions, and the program code into corresponding segments. Multiple segments can be linked to the address of one region, meaning they can be swapped at run-time by means of transferring another segment with its functions into a region. This leads to a behaviour known from caches: When a called function is not mapped to its region, there is an overlay miss and it has to loaded before execution can continue.

There is a toolchain available integrated into the GNU Compiler Collection that can automatically generate overlays for SPU applications [2].

[3] introduces a Code Overlay Generator (COG) that aims at producing a more optimal overlay mapping than the default IBM Cell SPU compiler. Their approach is based on a heuristic that works without constructing or solving an ILP, instead relying on a heuristic.

A more detailed report on optimisations for instruction SPM for average-case execution time optimisation is given in [13].
4 Cache Locking

Cache locking is a technique available in some systems that allow the operating system or application to control cache behaviour. Locking data in the cache, i.e. keeping it from being evicted, can be used to make memory access more predictable or even optimise the average execution time.

Since the approaches and algorithms targeted at optimising by cache locking are similar to those for Scratchpad Memory, I included this section.

4.1 Architectures

[12] lists Coldfire MCF5249, PowerPC 440, MPC5554, ARM 940 and ARM 946E-S as architectures that support cache locking.

Cell The PowerPC element of the Cell multiprocessor allows software to lock cache contents in place, which allows it to optimise memory access, among others, for fast predictable response times in hard real-time systems.

x86 Cache locking is not a supported feature of the x86 architecture, but still possible through processor-specific cache control mechanisms. While the use of those is discouraged in performance-critical scenarios, there are some applications that make use of this feature:

- **CAR** Cache as RAM is a technique employed by the coreboot open-source bios to increase the amount of memory available to the CPU before the initialisation of the RAM controller.

- **CARMA** Carma is a framework to establish a trusted computing base requiring only the CPU of a computer to be trusted. It is motivated by the fact that PCI allows hardware to access the RAM and memory mapped regions in other peripherals. This provided an entry point for malware nested in PCI devices (e.g. NIC firmware flash) as well as cold-boot attacks where sensitive information is gained by freezing RAM chips and thus being able to read their contents after removing them from the system. The technique employed by CARMA is based on the approach of CAR. But, instead of using L1 cache, they are mapping and locking a portion of RAM into the L2 cache, which gives it a piece of general-purpose memory instead of splitting instruction and data memory.

4.2 Cache Locking vs. Scratchpad

For example, in [12], Isabelle Puaut and Christophe Pais compare the effects of the instruction cache locking optimisation from [11] for both SPM and locked cache. The differences between the two mechanisms is hidden by a function `Load` that takes care of the following:

- **Cache Locking** When relying on cache locking, `Load` scans all the program lines of the basic block to load and checks if there are free cache lines in each set a line is mapped to. If there is available cache lines, it then locks the scanned content in cache. This approach requires little or no modification of the original memory layout.

- **Scratchpad** For SPM, `Load` uses a first-fit strategy to allocate an entire basic block. This has to be done at compile-time to determine the memory address the block will be executed at.

This is an example for the portability of algorithms between SPM and cache locking optimisation. The detailed results are given in Figure 6, while scratchpad memory is not generally better than cache in the given scenario, the WCET is not significantly higher.

Furthermore, the authors voice the following concerns:
Cache pollution The granularity of caching, meaning cache lines of fixed size, may lead to cache pollution, the consequence of which is that data is unintentionally locked into the cache because it’s located in the same line as data that is intentionally locked.

SPM fragmentation As with any memory management, scratchpad memory may become fragmented when exposed to continuous allocations and deallocations.

Cache pollution directly affects the WCET because the time needed to load and lock a cache block is longer than necessary when including data that is not needed.

Cache is able to handle large basic blocks without performance loss because it is working with the independent granularity of its own block size. SPM, on the other hand, is very susceptible to basic blocks that are to large for the available memory, which may be worsened by aforementioned fragmentation. An example is given in Figure 7, where the jfdctint benchmark shows the explained behaviour.

5 Scratchpad in real-time systems

When developing software for hard real-time it is crucial to prove that it can meet specified reaction times. These are proven by calculating the worst-case execution time, short WCET.

Naturally - it is in the interest of developers, and those who evaluate hard real-time systems, to give as tight an upper bound as possible.

Optimising for WCET differs from average-case execution time (ACET) optimisation because it has to rely on the formal calculation and guarantee of the execution time along the worst-case execution path (WCEP). Because of this, most of the algorithms geared towards reducing the ACET are not suitable for WCET reduction, especially those relying on profiling.
5.1 Caches

There are several aspects of cache behaviour that make it difficult to guarantee tight WCET boundaries. Having a processor with cache leads to unpredictable timing, since its internal state at a specific point of time is unknown to compiler and developer. This leads to WCET estimates being pessimistic, assuming cache misses wherever it’s not clear if a page is cached, or even having to ignore the cache entirely[9].

A further aspect of those uncertainties, timing anomalies, may lead to cache misses being better for the whole WCET than a cache hit, which further complicates giving tight boundaries[14].

Using Scratchpad Memory alleviates those problems - as well as providing a method to specifically optimise execution time of the worst case execution path.

5.2 WCET centric data allocation to scratchpad memory[15]

In [15], the authors analyse different approaches to lower the WCET of an application by statically allocating data to SPM. First, they formulate an ILP which, when solved, produces an optimal allocation of variables to the scratchpad. After that, there is an analysis of both knapsack and branch-and-bound approaches to solving the problem faster, as well as a greedy heuristic to be able to allocate larger amounts of data within reasonable time.

While the approaches given in this paper can be used to allocate variables on the stack, it is only possible for non-recursive ones that can be treated as global variables.

5.2.1 ILP

The basis of the ILP to optimise the WCET is a set of decisions $S_v \in \{0, 1\}$ that determines for each variable $v \in \text{allvars}$ whether it is allocated to SPM($S_v = 1$) or conventional memory($S_v = 0$). To ensure that all the allocated variables actually fit in the SPM, there is the constraint

$$\sum_{v \in \text{allvars}} S_v \cdot \text{area}_v \leq \text{scratchpad.size}$$

that ensures that the sum of the area needed for each allocated variable $\text{area}_v$ is lower than the size of the SPM $\text{scratchpad.size}$.

Calculation of the WCET of a loop is done through analysis of the directed acyclic graph(DAG) representing its control flow. It is assumed that the DAG of each loop has exactly one source and one sink node, the latter may be a virtual sink node when none is given.

The WCET of the DAG rooted in each basic block $i$ is called $W_i$. It is calculated from the sink node in the DAG to the source node, the first being

$$W_{\text{sink}} = \text{cost}_{\text{sink}} - \sum_{v \in \text{allvars}} S_v \cdot \text{gain}_v \cdot n_{i,sink}$$

where $\text{cost}_i$ is the execution time of the basic block $i$ and the reduction of which is calculated by multiplying the gain in access time of each variable $\text{gain}_v$ through its allocation to scratchpad with the number of occurrences $n_{i,sink}$.

Similarly for every edge in the DAG $i \rightarrow j$:

$$W_i \geq W_j + (\text{cost}_i - \sum_{v \in \text{allvars}} S_v \cdot \text{gain}_v \cdot n_{i,i})$$

Which gives us the WCET of the loop body as $W_{\text{source}}$. The WCET of the whole program can be determined by multiplying the WCET of each loop with the a known constant $lb$, which is the known
maximum number of iterations. This gives us the maximum execution time cost of the innermost loops, which can now be used to construct the same constraints for the next level of loop nesting until the WCET for $W_{entry}$, the unique entry node of the program, is known. Thus, the objective function of the ILP is the WCET of the entire program: $W_{entry}$

5.2.2 Knapsack?

While formulating memory allocation as a knapsack problem may be an intuitive choice, it is not appropriate for WCET-optimising SPM allocation.

The problem is that the reduction achieved through the allocation of variables to scratchpad is not additive. This is because, whenever one path is optimised, it might become faster than another path, leading to that being the new worst-case execution path, which lessens the achieved reduction in worst-case execution time. A graphical example of changing WCEPs is given in figure 15 in section 5.5.

Since the WCET reduction achieved through the allocation of each variable is heavily dependent on the allocation of other variables, knapsack cannot be used to solve the allocation problem in this instance.

5.2.3 Branch and Bound

Branch and bound is the improved approach chosen to find a perfect solution to the ILP given in 5.2.1. To not be forced to try all the possible combinations of variable allocations, branch and bound is an approach that utilises a heuristic to discard large amounts of possible solutions without calculating the result of all of them.

This is achieved by representing the decision of the allocation of each variable as a layer in a tree structure. While traversing the tree, after each WCET that is lower than the WCETs found before, all those subtrees not yet traversed are discarded for which a heuristic shows that their lowest possible WCET is above the minimum already found.

See figure 8 for a graphical example.

![Figure 8: Pruning the branch-and-bound search tree](image)

A good heuristic for the upper bound of a subtree, referred to as $UB()$ from now on, is essential for the branch-and-bound approach to be able to cut away large portions of the decision tree. This is essential to lessen the computational effort required for finding an optimal solution.
While it is not reliable to optimise the worst-case execution time, knapsack optimisation provides a definite upper bound for the possible WCET reduction through variable allocation. It is solved through dynamic programming, and requires the following parameters:

1. Variables $v_i$ to allocate.
2. Size of each variable $area_{v_i}$ to allocate.
3. Limit given through the size of the SPM: $scratchpad_{size}$
4. Maximum possible execution time reduction through allocation of variables given by the maximum reduction achieved on any execution path: $bound_v$

The effort necessary is still exponential, which is why even branch-and-bound is not feasible for larger programs and large SPM.

5.2.4 Greedy Heuristic

To provide a feasible solution to the problem of static data allocation to SPM, there is a greedy heuristic given in [15] that has a significantly lower complexity. The results of a comparison given in [10] indicate that the given greedy heuristic is close to the optimal case when considering the achieved WCET reduction.

[15] incorporate unfeasible path detection to further reduce their WCET. They detect unfeasible paths by searching for conflicting assignments that are of the form $variable := constant$ and conditional branches of the form $variable_{relational}operatorconstant$. For further information, the reader is referred to [4].

5.3 Optimal Static WCET-aware Scratchpad Allocation of Program Code [5]

While the previous paper focused on data allocation to SPM, [5], as the title indicates, focuses on the allocation of program code.
Figure 10: WCET reduction through optimisation with original ILP, branch-and-bound and greedy heuristic for various applications\[15\]

5.3.1 ILP

The ILP defined is very similar to the one given in \[15\], which is why I will narrow the explanation down to the differences between the two. First, program code is usually worked with on the granularity of basic blocks, but defining decision variables for those makes no difference in the general ILP layout.

A more important consideration done in \[15\] is the size and speed penalty of jump and branch instructions. Encoding limits the address “distance” an instruction can jump or branch over, which, together with SPM being mapped to a different address space than the RAM leads to more instructions being...
required when there is a transfer from basic block to basic block across the two types of memory.

Figure 11: Possible jump scenarios

$x_i$, $x_j$ and $x_k$ are the decision variables for the basic blocks $b_i$, $b_j$ and $b_k$. There are three scenarios to transfer control between basic blocks on typical embedded processors:

Implicit jumps transfer control between consecutive blocks without a jump or branch instruction, when the end of one basic block is reached and the next one begins. The execution time penalty for placing basic blocks connected by implicit jumps in different memory spaces can be modelled by:

$$jP_{impl}^i = (x_i \otimes x_j) \cdot P_{high}$$

$\otimes$ represents the logical XOR, $P_{high}$ is the jump penalty for jumping across memory spaces.

Unconditional jumps are penalised when the basic blocks lie in different memory spaces. When they share the same, there is the much smaller penalty $P_{low}$. The basic blocks may even be consecutive in the memory they are allocated to, which happens when all in-between blocks are allocated differently, and causes a penalty of 0. The general penalty caused by unconditional jumps from block $b_i$ to $b_j$ is defined as:

$$jP_{uncond}^i = (x_i \otimes x_j) \cdot P_{high} + (x_i \otimes x_j) \cdot (1 - \prod_{b_k \in Figure 11} (x_i \otimes x_k)) \cdot P_{low}$$

A conditional jump is regarded as a combination of both implicit and unconditional jump, it’s penalty being:

$$jP_{cond}^i = (x_i \otimes x_j) \cdot P_{high} + (x_i \otimes x_j)^{cond} \cdot P_{high} + (x_i \otimes x_j) \cdot (1 - \prod_{b_k \in Figure 11} (x_i \otimes x_k)) \cdot P_{low}$$

Those three are then integrated into the ILP, being added to the WCET at every edge in the directed acyclic graph that represents the given code.

Aside from those performance impacts, the memory consumption of additional jump instructions has to be considered when allocating code to SPM. [15] gives the following size penalty for the different jump conditions:

$$s_i = \begin{cases} 
(x_i \land \neg x_j) \cdot S_{impl} & \text{if JS of } b_i \text{ is implicit} \\
(x_i \land \neg x_j) \cdot S_{uncond} & \text{if JS of } b_i \text{ is unconditional.} \\
(x_i \land \neg x_k) \cdot S_{impl} + (x_i \land \neg x_j) \cdot S_{uncond} & \text{if JS of } b_i \text{ is conditional.} \\
0 & \text{else}
\end{cases}$$

5.3.2 Compiler

To use the given ILP, [15] used the architecture of their WCET-aware C compiler WCC. ILP-based WCET-aware SPM code allocation is done after all other optimisations. The entire architecture is outlined in figure [12].
5.3.3 Evaluation

For evaluation, [15] uses 73 different real-life benchmarks, the result for some of which is given in figure [13]. The size of the scratchpad is 48kb, 47 of which are usable after reserving 1b for system code. Program code size of all benchmarks are between 52 bytes and 18kB, so all of them fit into the SPM. Each of the five benchmarks was run with the scratchpad size being restricted to 10%, 20% etc. up to 100% of the program size.


5.4.1 Definition of problem

[11] provides WCET optimisation through dynamic cache locking. Puaut first gives a greedy algorithm to find near-optimal solutions that are then used as an initial population for a genetic algorithm which further optimises the result.

Unlike the previously discussed static SPM allocations, dynamic SPM or cache locking changes the allocation of variables or basic blocks during program execution. The dynamic allocation of can be split into two problems: First, the choice of reload points at which new content gets locked in the cache and old content may be evicted. Second, the data to load at these reload points has to be determined.
5.4.2 Choice of reload points

While reload points (RP) could be placed at every instruction in the program, the vast number of possibilities would cause an enormous complexity. In [11], the author chooses to limit the placement of reload points to natural locations, which are the headers of functions and loops.

To limit the amount of reload points, the user may specify `max_reload_points`.

To decide whether a RP is worth choosing, an estimate of the possible WCET reduction is calculated. For those calculations it is necessary to give an overview of the cache model used: The instruction cache is W-way set associative. It contains $B$ blocks of $S_B$ byte each, with the total size being $S_C = B \times S_B$. Because the instruction size of the CPU is fixed, each cache block can hold exactly $ipcl$ instructions.

There are several constants to describe timing behaviour of the cache:

1. $t_{hit}$ and $t_{miss}$ are the latencies caused by cache hits and misses.
2. $t_i$ and $t_l$ are caused by loading and locking the cache respectively.

The number of cache misses required for load and lock of $pl$ instruction lines is $t_i + t_l \times pl$

To evaluate whether a reload point is worth being placed before the loop $L$, the greedy algorithm uses the Cost function $CF(L)$ given below. Furthermore, it relies on a function $f(bb)$ or $f(pl)$ that gives the number of executions of a basic block $bb$ or instruction line $pl$ on the WCEP of the loop $L$. While $pl(L)$ returns all instruction lines of the loop, $mfpl(L)$ returns all the most executed instructions that fit into the cache.

$$WCET_{cache}(L) = \sum_{pl_i \in pl(L)} f(pl_i) \times (t_{miss} + (ipcl - 1) \times t_{hit})$$

$$WCET_{locked}(L) = \sum_{pl_i \in mfpl(L)} f(pl_i) \times ipcl \times t_{hit}$$
$$\quad + \sum_{pl_i \in pl(L) - mfpl(L)} f(pl_i) \times ipcl \times t_{miss}$$
$$\quad + \sum_{ph \in pre\_head(L)} f(ph) \times (t_i + t_l \times |mfpl(L)|)$$

$$CF(L) = WCET_{cache}(L) - WCET_{locked}(L)$$

$CF(L)$ indicates the benefit of placing a reload point in the pre-header of loop $L$. If $CF(L)$ is positive, the WCET of the loop $L$ is expected to be lower when cache content is locked in the reload point preceding it.

The greedy algorithm sorts the reload points and selects the `max_reload_points` first.

5.4.3 Choice of content to load&lock

For each RP that is decided to be worth using, the benefit of the load&lock of each basic block (BB) following that RP is considered. Similarly to the benefit of RP, this is done through a formula that weighs the estimated WCET using cache in regular LRU mode against that achieved while locking the BB to said cache. The actual algorithm defined in [11] for this step is given in [12]. It chooses the $N$ most beneficial basic blocks, iterates the reload points and loads all program lines of each basic block at all the reload points that precede it. This is done iteratively until there are no more beneficial basic blocks or the WCET after an iteration is worse than that of the iteration before. A lower value for $N$ means that the WCEP and WCET are reevaluated more often, which increases the effect of the optimisation.
Figure 14: Selection of cache contents

The choice of beneficial basic blocks (SelectMostBeneficialBB) is, like the choice of reload points, done through evaluation of a cost function:

\[ WCET_{cache}(bb, L) = \sum_{pl_i \in PL(bb)} f(pl_i) \cdot (t_{miss} + (ipcl - 1) * t_{hit}) \]

\[ WCET_{nocache}(bb, L) = \sum_{pl_i \in PL(bb)} f(pl_i) \cdot ipcl * t_{miss} \]

\[ WCET_{locked}(bb, L) = \sum_{pl_i \in PL(bb)} f(pl_i) \cdot ipcl * t_{hit} \]

\[ + \sum_{ph \in pre-head(L)} f(ph) \cdot (t_i * \frac{B}{|bb|} + t_i * |bb|) \]

\[ CF(bb, L) = (WCET_{locked}(bb, L) - WCET_{cache}(bb, L)) \]

\[ + (WCET_{locked}(bb, L) - WCET_{nocache}(bb, L)) \]

5.4.4 Example

Figure 15: Example for the greedy algorithm given in [11]

An example for two iterations of the greedy algorithm is given in figure [15]. It shows the behaviour
previously discussed in section 5.2.2: Allocation of contents on the WCEP to either SPM or locked cache may change the WCEP, leading to less reduction in WCET than on the original path, which is not additive.

5.4.5 Implementation details

Instead of restructuring the code or executable of the given application, one approach for inserting reload points given in [11] relies on the debug function of the processor platform being used. A breakpoint is used at every reload point, the exception caused by which is captured by the processor and handled by an external manager that takes care of the cache locking.

5.4.6 Genetic Algorithm

A genetic algorithm uses Darwin’s theory of evolution to find solutions to a problem. It does so by evaluating the fitness of individuals in a pool of specimens and applying evolutionary mechanisms, e.g., crossover and mutation. There is no guarantee that a evolutionary algorithm finds a "good" solution, and the time it takes for a random initial population to become reasonably fit is too long. Because of this, the genetic algorithm is used to increase the fitness of the results of the greedy heuristic given before.

The formal definition of the parameters for said algorithm is as follows:

Codification (Representation of an individual) Individuals are represented by "chromosomes", which are arrays of tuples of the form 

\[(rp, contents)\]

\(rp\) identifies a reload point and contents the cache contents to be locked at that point.

Fitness The fitness of an individual is the WCET it achieves.

Selection The probability of the selection of one individual is linearly dependent on its WCET.

Crossover and mutation Crossover is done by randomly selecting a point in the chromosome, everything before which comes from one parent, everything after from the other. There are three mutation mechanisms implemented: \(M_{rem}\) removes one randomly selected reload point, \(M_{add}\) adds a random reload point and \(M_{chg}\) randomly changes the content of one reload point.

5.5 Evaluation

<table>
<thead>
<tr>
<th>Appli</th>
<th>Cache LRU</th>
<th>Cache PRR</th>
<th>Greedy</th>
<th>Genetic</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>des</td>
<td>7.9</td>
<td>17.6</td>
<td>16.3</td>
<td>14.1</td>
<td>323</td>
</tr>
<tr>
<td>adpcm</td>
<td>7.8</td>
<td>8.2</td>
<td>30.4</td>
<td>30.2</td>
<td>300</td>
</tr>
<tr>
<td>minver</td>
<td>7.8</td>
<td>17.9</td>
<td>39.5</td>
<td>24.1</td>
<td>181</td>
</tr>
<tr>
<td>fft</td>
<td>8.3</td>
<td>24.7</td>
<td>15.6</td>
<td>11.8</td>
<td>217</td>
</tr>
</tbody>
</table>

Figure 16: Performance results given in [11], miss ratio of LRU, PPR, the heuristic and genetic algorithm.

To evaluate the performance of both the greedy heuristic and the genetic algorithm, Puaut compares the hit to miss ratio the Heptane open source cache-aware WCET estimation tool yields for compiled MIPS R2000/R3000 binary code. They are compared to regular cache using the cache replacement strategies LRU and pseudo round-robin (PPR), the latter of which is chosen because it is a hard to predict strategy. Figure [16] shows that neither the results of the greedy algorithm nor their genetically optimised versions yield a better miss ratio than LRU. There are, however, cases in which the hard to predict PRR
strategy performs worse, which indicates that hard to predict cache replacement strategies could be an application for the algorithms proposed in \[11\].

6 Conclusion

The basic message every paper on SPM that I read carries is: SPM is able to compete with regular cache, not seldom overcoming the efficiency cache can offer. Development on optimising compilers continues making advancements, so it should be only a matter of time before optimisations for explicitly managed memory hierarchies are implemented outside of research compilers.

While general-purpose systems require more work to make existing applications work with SPM, embedded systems have the benefit of having software that is often tailored to, or at least specifically compiled for the specific architecture. Together with increased energy efficiency and, compared to cache, decreased complexity, this makes SPM a perfect mechanism to optimise embedded systems.

Even tho most of the interesting optimisations for general-purpose and high-performance multicore systems are explained in the paper of my colleague Axel Ratzke[13], there are conclusions I can draw from my research on the known applications of scratchpad memory and explicitly managed memory hierarchies: The success of GPGPU programming for high-performance and especially high-efficiency calculations and the success of the Cell architecture indicate that the concept of Scratchpad memory is a worthwhile and important topic for research on optimisation of those applications.

Mobile devices, specifically Android smartphones with their java-based portable applications, would require a JIT to be able to optimise software for SPM, which is something I would like to see being researched.

In conclusion: Being able to automatically compile and optimise code to properly use explicitly managed memory hierarchies appears to be an important step to increase the efficiency of computing applications.

7 Bibliography

References


[16] Lehrstuhl Informatik Xii, Rajeshwari Banakar, Rajeshwari Banakar, Stefan Steinke, Stefan Steinke, Bo sik Lee, Bo sik Lee, M. Balakrishnan, M. Balakrishnan, Peter Marwedel & Peter Marwedel (2001): Comparison of Cache- and Scratch-Pad based Memory Systems with respect to Performance, Area and Energy Consumption.