HARDWARE MANAGED SCRATCHPAD MEMORY FOR EMBEDDED SYSTEMS

Ben Rudzyn (3021008)
CONTENTS

Introduction p. 2
Approach p. 3
Software simulation p. 4
Software modification p. 5
Experimental results p. 6
Performance analysis p. 8
Limitations p. 11
Further analysis p. 13
Future work p. 14
Conclusion p. 15
References p. 16
Author’s comments p. 17
INTRODUCTION

Designers of embedded systems continually endeavour to obtain the maximum performance from their product with the least power possible. In general purpose processors, the instruction memory system often accounts for a large portion of overall energy consumption. In these systems, an instruction cache is commonly employed between the CPU and main memory, and generally reduces execution time and energy expenditure, despite an often complex tag system. Yet in embedded systems, on chip caches can still consume from 25 to 45% of total chip power [Banakar et al, 2002]. However, embedded systems differ from general purpose systems in that only a finite set of programs is run on the system. In this case it is possible to completely examine and profile the programs to be executed, and thus either design the embedded system around the application, or modify the application to best utilise the system.

A scratchpad memory (called SPM hereafter) is an array of SRAM cells, with only simple address decoding. It does not store any extra tag information that would be found in a cache, such as a dirty or valid bit. In the traditional memory hierarchy, the CPU only has access to the cache, which in turn accesses main memory (DRAM). In the SPM configuration, proposed by Janapsatya [Janapsatya et al, 2004], an instruction scratchpad memory is located inside the address space, and there is no instruction cache. The CPU thus directly accesses either main memory or the scratchpad memory, and data can be transferred from the main memory to the SPM via a separate bus, as seen in the diagram below.

![Diagram of memory hierarchy](image)

To use the SPM configuration effectively, the compiled code is initially loaded into DRAM. Whenever a repeatedly used section of code is encountered, it is copied into the SPM, and the segment proceeds to be executed from the much faster SPM. Other sections of code which are run a few times, or run sporadically, can be executed directly from DRAM. By removing the instruction cache and directly accessing main memory, extra cache address decoding and tag checking is avoided. This saves on execution time and energy consumption, and is particularly advantageous for those instructions in a program which do not exhibit temporal locality. The benefits of the cache system are still maintained (specifically the exploitation of locality) by the runtime copying of blocks of code from the main memory to the SPM. These blocks are then run directly from the Scratchpad, without any extra tag comparisons (or storage).

The motivation for using a SPM instead of a cache can be seen in this simple example. If we model part of a programs execution as a directed graph (vertices are basic blocks, edges are transitions, weights are the number of transitions found through profiling), then it is known that execution of block B will always follow block A. Hence B can always be loaded into the SPM when A is, eliminating the need for extra tag comparisons and any potential misses.
To manage the SPM, and to control the runtime copying of instructions from main memory into the SPM, a custom hardware controller is needed. It is activated and controlled by instructions placed within the program prior to execution. A new instruction, designated a Scratchpad Managing Instruction (smi), is added to the ISA for this purpose.

The Scratchpad Memory Controller (referred to as the SPC or Controller hereafter) contains three components:

- Control Logic – activates or stalls the other components in the SPC in the correct order
- Basic Block Table – a lookup table that is initialised before program execution. It stores all the information regarding memory transfers between DRAM and the SPM in a DMA style (ie starting address of instructions in DRAM, starting address of transfer to SRAM, number of consecutive instructions to copy)
- Memory Controller – this component controls the timing of signals to transfer data between DRAM and the SPM.

The strategy for using the SPM is as follows: at the start of program execution, instructions are fetched directly from DRAM. When an smi is decoded, the CPU will activate the SPC and stalls. Once the SPC is activated, it starts copying instructions from DRAM into SPM (the smi instruction specifies which entry in the Basic Block Table is to be executed). Once copying is complete, the SPC reactivates the CPU and sleeps itself. The CPU will continue executing instructions from the DRAM until a jump instruction directs it into the SPM. It will then continue executing as normal, unaware that the instructions are coming from the SPM. Again, on execution of another jump, the CPU will return to fetching from DRAM.

The methodology and initial design for the memory modifications were provided through research by Andhi Janapsatya [Janapsatya et al, 2004]. A hardware model of the Scratchpad Memory Controller was designed and implemented during a summer project undertaken by myself.
SOFTWARE SIMULATION

Unfortunately, it proved too difficult and time consuming to obtain usable results through hardware simulation. Consequently a software simulator was developed to test the methodology. A previously modified version of sim-fast was used, which supports the syscalls of the SS CPU (ie all syscall code is stored inside the internal SS memory, rather than making a call to an external OS). This modified version of SimpleScalar was provided by Jorgen Peddersen. Further alterations were then made to accommodate the SPC, a SPM, and the modified DRAM needed for experimentation. The details of the final simulator are presented below, as too is a very simplified diagram outlining the structure of the hardware model being simulated.

<table>
<thead>
<tr>
<th>CPU</th>
<th>6 stage, statically scheduled, single instruction pipeline</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functional units</td>
<td>1 integer ALU, 1 integer multiplier, 1 integer divider</td>
</tr>
<tr>
<td>Functional unit latencies</td>
<td>All single cycle</td>
</tr>
<tr>
<td>Instruction cache</td>
<td>2 Kb (256 entries), direct mapped, 1 word line size (64 bit), 1 cycle hit latency, 2 cycle miss latency (plus memory delay)</td>
</tr>
<tr>
<td>Instruction SPM</td>
<td>8 Kb (1024 entries), 1 cycle hit latency</td>
</tr>
<tr>
<td>Instruction main memory</td>
<td>8 Mb SDRAM (10ns), simplified burst mode 10-1-1-1*, 4 word line size</td>
</tr>
<tr>
<td>Data cache</td>
<td>2 Kb (512 entries), direct mapped, 1 word line size (32 bit), 1 cycle hit latency, 2 cycle miss latency (plus memory delay)</td>
</tr>
<tr>
<td>Data main memory</td>
<td>8 Mb SDRAM (10ns), simplified burst mode 10-1-1-1*, 4 word line size</td>
</tr>
</tbody>
</table>

* simplified burst mode takes 10 cycles the first time a line is accessed, and 1 cycle for every other access to that same line (read or write), unless another line is accessed in between. It does NOT automatically load all 4 words into cache/SPM, but stores the line in a local buffer.

Simulation results for the software simulator were compared to previously obtained hardware simulation results. For small programs (< 50,000 cycles), accuracy was within 2%. However, for larger programs, such as the benchmarks (> 1 million cycles), accuracy increased to within 0.05% of hardware results. The main discrepancy lay with calculating delays during branches, particularly when a branch and jump delay overlapped each other in the same cycle.
SOFTWARE MODIFICATION

The new instruction, \textit{smi}, contains a single 16 bit operand, which represents the address of a Basic Block Table (BBT) entry for that instruction. This allows up to 65536 individual \textit{smi}'s to be inserted into a program if the entire BBT is utilised.

The following code example illustrates how the \textit{smi}'s were inserted into the code to make use of the SPM. The code on the left represents a small \textit{for} loop, which starts at $\text{L295}$ and finishes at the \textit{bne} after $\text{L294}$. In order to copy this small loop into the SPM, an \textit{smi} is inserted, which runs entry 0x12 in the BBT of the Controller. This entry tells the Controller to copy 13 instructions from DRAM (starting at the address of $\text{SL295}$) to address 0x0f000118 (address 0x118 in the SPM). An extra \textit{jump} is then inserted to direct program execution into the SPM, and another \textit{jump} is placed at the end of the loop to return execution to main memory once the loop is completed (note that the second \textit{jump} also needs to be copied into the SPM). In this example, all internal \textit{branches} are relative, and so these do not have to be updated. When there are \textit{jump}'s inside the loop block, then the absolute address would need updating as well.

```
li      $4,     0x3b9aca00
move    $16,$0
$L295:
sll     $2,$16,2
addu    $2,$2,$17
.set    noreorder
lw      $3,0($2)
#nop
set    reorder
beq     $3,$0,$L294
slt     $2,$4,$3
bne     $2,$0,$L294
move    $4,$3
move    $7,$16
$L294:
addu    $16,$16,1
slt     $2,$16,257
bne     $2,$0,$L295
bltz    $7,$L284
sll     $5,$8,2
addu    $6,$5,$17

$end12:
bltz    $7,$L284
sll     $5,$8,2
addu    $6,$5,$17
```

Loop detection software, as well as a hot function detector, both provided by Seng Lin Shee, were used to identify the key loops that would be transferred into the SPM. These two programs enabled the identification of all loops that accounted for more than 0.1% of execution time (ie time spent in loop wrt total execution time). However, this software was not able to identify basic blocks that might be beneficial to copy into the SPM. Thus for this experiment, the only blocks copied into the SPM were loops (\textit{for} or \textit{while}), rather than following the strict definition of a basic block. Furthermore, this means that large blocks of code that do not loop often could not be identified or copied into the SPM.

The code was modified by hand, as too was the updating of the BBT in the Controller. The correct usage of the SPM was also monitored by hand.
EXPERIMENTAL RESULTS

Four separate cases were tested for comparison. Note that in each case, the data memory and data cache remain unchanged (only changes to the instruction memory system are being considered here). The structure of each test case was as follows:

- **mem** – this provided a baseline set of results in which only main memory (DRAM) was used. There was no cache or SPM employed here.
- **cache** – these results represent the standard cache configuration, and the results that would be seen with simple traditional embedded systems.
- **spm** – in this case the cache is removed and replaced with a SPM and corresponding SPC.
- **optimised** – the hardware layout of this case is the same as in the spm case above. However, the software placement of *smi* instructions is slightly different. All *smi*’s are placed at the start of the program, rather than directly before each loop. This case was tested to identify the effect that repeated calls to the SPC would have on overall performance (more on this later).

A subset of the raw results are displayed on the following page. Below are plots of the two key indicators, CPI and % speedup from memory (ie speedup from using only main memory).

![CPI for benchmarks](image)

![Speedup from mem only](image)
<table>
<thead>
<tr>
<th></th>
<th>jpeg</th>
<th>adpcm</th>
<th>pegwit</th>
<th>g721</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>cjjpeg</td>
<td>djjpeg</td>
<td>rawcaudio</td>
<td>rawdaudio</td>
</tr>
<tr>
<td>mem</td>
<td># instr</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>clock cycles (cc)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>76,894,449</td>
<td>23,415,532</td>
<td>43,563,721</td>
<td>38,198,826</td>
</tr>
<tr>
<td>cache</td>
<td># instr</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>cc</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>cache misses</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1,619,656</td>
<td>523,221</td>
<td>13,540</td>
<td>11,155</td>
</tr>
<tr>
<td>spm</td>
<td># instr</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>16,430,820</td>
<td>5,083,491</td>
<td>9,429,298</td>
<td>7,587,780</td>
</tr>
<tr>
<td></td>
<td>cc</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>57,977,611</td>
<td>18,369,116</td>
<td>23,177,124</td>
<td>21,557,162</td>
</tr>
<tr>
<td></td>
<td># smi calls</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1,619,656</td>
<td>523,221</td>
<td>13,540</td>
<td>11,155</td>
</tr>
<tr>
<td></td>
<td>instr copied to SPM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>57,977,611</td>
<td>18,369,116</td>
<td>23,177,124</td>
<td>21,557,162</td>
</tr>
<tr>
<td></td>
<td>instr run from SPM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>57,977,611</td>
<td>18,369,116</td>
<td>23,177,124</td>
<td>21,557,162</td>
</tr>
<tr>
<td></td>
<td>% total instr run from SPM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>56.1</td>
<td>73.7</td>
<td>70.7</td>
<td>72.4</td>
</tr>
<tr>
<td>optimised</td>
<td># instr</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>16,422,026</td>
<td>5,081,339</td>
<td>9,429,151</td>
<td>7,587,633</td>
</tr>
<tr>
<td></td>
<td>cc</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>55,144,597</td>
<td>15,854,510</td>
<td>23,120,060</td>
<td>21,509,131</td>
</tr>
<tr>
<td></td>
<td># smi calls</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>16,422,026</td>
<td>5,081,339</td>
<td>9,429,151</td>
<td>7,587,633</td>
</tr>
<tr>
<td></td>
<td>instr copied to SPM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>55,144,597</td>
<td>15,854,510</td>
<td>23,120,060</td>
<td>21,509,131</td>
</tr>
<tr>
<td></td>
<td>instr run from SPM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>55,144,597</td>
<td>15,854,510</td>
<td>23,120,060</td>
<td>21,509,131</td>
</tr>
<tr>
<td></td>
<td>% total instr run from SPM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>56.1</td>
<td>73.7</td>
<td>70.8</td>
<td>72.4</td>
</tr>
</tbody>
</table>
PERFORMANCE ANALYSIS

The results indicate that in terms of clock cycles, use of the scratchpad memory does not outperform that of a cache. Only two tests (pegwitenc, pegwitdec) found that the SPM was faster than the cache, and this was only in the optimised case! In another two tests (g721enc, g721dec), the use of a scratchpad only improved performance by ~3% and ~7% (spm, optimised). These results are not very promising.

A deeper analysis of the benchmarks themselves proves insightful as to why performance was so lacking, and highlights particular limitations or improvements that need to be addressed with the SPM configuration.

pegwit

Instructions per loop copied into the SPM: avg 25, min 7, max 40

The most common loops are identical for each of the three pegwit programs, although the frequency with which each loop is run differs (eg one loop runs 41466 times for pegwitenc, and only 21116 for pegwitdec). However, there is quite an obvious contrast between the performance boost of using a SPM for the three tests.

One major issue is the small number of iterations of each loop. For one loop in particular (ec_field:376), although it is called 21118 times, it is only ever iterated 1, 2 or 3 times. Since this loop only has 7 instructions (in assembler), an additional 2 instructions (the extra jumps) represents an almost 30% increase in code size when it is only iterated once. This is one of the main reasons why the SPM does not outperform the cache in pegwitkey.

Another obvious hindrance is the overhead of running the SPC and copying instructions into the SPM each time they need to be used. In the case of pegwitkey, there is only a 5.3% miss rate for the cache, which implies that most instructions are only ever loaded into the cache once. However, the SPM will be reloaded each time a loop is called, adding a substantial overhead delay. This extra copy overhead is removed in the optimised case, and thus the performance of the optimised case is much closer to that of the cache.

SPM performance actually overtakes cache performance for pegwitenc and pegwitdec. While virtually the same instructions are copied into the SPM as in the pegwitkey test, the pattern of program execution is very different. For these two tests, cache miss rates increase to 25.2% and 28.3% respectively, indicating that a much larger subset of the pegwit programs functions are being called. Having to fill the cache more often is comparable to having to fill the SPM each time the loop is called, and thus the overhead problem for pegwitkey doesn’t seem to have as great a comparative impact here. In the optimised case however, where the SPM is filled once at the start of the program, while the cache continues to need refilling, the SPM always has a hit (at least for the specific loops of interest). Thus despite having to execute two more instructions each time the loop is encountered, performance is better in the optimised case than for the cache case.

The results here imply that a SPM is most beneficial for programs whose execution jumps around a lot (ie high cache thrashing).
g721

Instructions per loop copied into the SPM: avg 17, min 7, max 32

Similar to the pegwit tests mentioned above, cache misses for these tests are very high, sitting at 33.4% and 32.3% for encode and decode respectively. This implies that the program runs a large number of instructions sequentially, rather than reducing most of the processing to a series of loops (i.e., the entire program can be thought of as one big loop). However, this time the SPM provides quite poor results.

Unfortunately, g721 is an extreme case of the ‘one big loop’ problem. While there are only ~1900 raw instructions in total, and out of this only 3 key loops (each running 147520 times), these loops only account for 10.11% (enc) and 9.59% (dec) of total run time. The g721 program is broken down into a large number of very small routines, so instead of calling one large function that loops many times over the input data, a small series of functions are called sequentially instead. This explains the large percentage of cache misses. It is made worse in that the three most commonly run loops are only iterated 6 times on each call.

In the optimised case, the speedup is more than doubled when compared to the spm case (3.1% vs 7.1%). However, since the loops only account for 9.9% of total runtime, the small number of instructions copied into the SPM are not enough to cause a large improvement.

adpcm

Instructions per loop copied into the SPM: 62 rawcaudio, 51 rawdaudio

In both programs, only one giant loop was copied into the SPM. This is the virtual opposite style of program to that of g721. The entire program consists of a main function, and a processing loop. Cache misses are extremely low (0.1%), and yet the performance of the spm is close to that of the cache (again shows how crucial the code style is in the effectiveness of the SPM). The entire program is actually small enough to fit completely inside the SPM, although only the main loop was actually copied into it.

Another important aspect is that this main loop is only called 148 times, but iterated 520 to 1000 times, which means that the extra 2 instructions added to the block have very little impact. What does have an impact though, is the structure of the loop. In both cases, the loop contains 8 if/else clauses. This means that each time the loop is run, all 62 instructions (for rawcaudio) are copied into the SPM, even though only a fraction of these will be run with each iteration. However, since the loop is iterated so many times, this additional overhead of a couple of instructions again has little impact.

Similarly, there is very little advantage in the optimised case here. Because the loop is called so few times, but iterated so many, the overhead of copying the instructions into the SPM each time, rather than copying it only once, is not a big factor. In the optimised case, there is only a saving of ~50,000 cycles out of 21 million.

These two tests seem to be perfectly suited to a SPM. They only have one large loop, iterated over 500 times with each call, and the overhead of running the SPC and adding extra jump instructions is virtually nil. The spm case runs just as well as the optimised case. So why then is there not a closer match between the cache and spm results? The answer lies with library functions. The main compression or decompression loop only accounts for 74.9% (rawcaudio) and 68.5% (rawdaudio) of total execution time. The remaining time is spent reading or writing (literally using the read and write functions) between the external data file and the internal buffer. These functions were not copied into the SPM and thus could not benefit from the speedup of using SRAM cells.
jpeg

Instructions per loop copied into the SPM (cjpeg): avg 42, min 4, max 155
Instructions per loop copied into the SPM (djpeg): avg 107, min 10, max 216

Both of these programs had an eclectic mix of large and small loops that were copied into the SPM. There are a few interesting features that need to be mentioned. These program features were also seen in some of the other benchmarks, but not in such a great extent.

A large portion of the for loops consisted of a single line, which was generally an if statement. While each of these loops were iterated 257 times, this does not indicate how many times the inner if statement was processed. This represents a potentially large overhead, depending on the data file being operated on.

The largest overhead is caused by multiple nested loops. When a for loop is nested inside another for loop inside another for or while loop (occurs at least 3 separate times), the code profiler identifies the inner most loop as that which runs the most. This inner loop is consequently copied into the SPM to run faster. Each time the inner loop is run, the SPC is called to copy instructions into the SPM. Even though the cost of an individual SPC call compared to the number of times the inner loop is run might be small, this still represents a substantial overhead and waste of execution time. The SPC is called for each iteration of the middle loop, for each iteration of the outer loop, even though the instructions are already stored there. In contrast, a cache would not have to be refilled each time the inner inner loop was called.

Overhead due to the extra 2 jump instructions added into the code has generally been suppressed by the large number of iterations of each loop. However, particularly in the jpeg tests, there are a few examples of small loops (4-10 instructions), which may contain if statements, that iterate less than 4 times. For a loop of size 4, the overhead of an additional 2 instructions is 50% when it is only iterated once.

Even worse are the loops which exhibit fluctuating iterations (based on the data). An example is the loop found in jchuff.c (line 427). It’s size is 4 instructions, and is called 1502 times over the life of the program. However, the amount of times it iterates fluctuates from 1 to 7. It is inefficient to copy this loop to the SPM when it only iterates once, but is acceptable for an iteration of 7. Since there was no way to gauge how many iterations would be processed on a given activation of the loop, the loop was copied to the SPM each time it was called, even with the potentially large overhead.

All three of these issues (if inside for, loop in loop, fluctuating iterations) are avoided in the optimised case, which was one of the main reasons for testing this case separately. As a result, performance for the optimised case improved over the spm case for both tests.
LIMITATIONS

Analysis of the results above has highlighted quite a number of limitations to both the design, and the experimental testing of the scratchpad memory configuration. These are summarised and commented on in this section.

The size and structure of the loops being copied in the SPM plays an important role in the amount of overhead that the SPM generates. The following example shows how many clock cycles are needed to run a simple 8 instruction loop for two iterations. On comparison, it seems faster to load instructions into the SPM and run them from there, rather than having to load the cache (due to the initial miss penalty). However, in many cases there is a hit in the cache, while the scratchpad would be refilled again. In the optimised case, where the SPM is loaded only at the start, then it matches the performance of the cache (since the initial load of the SPM is amortised over the life of that loop).

<table>
<thead>
<tr>
<th>Cache (initial miss)</th>
<th>First iteration</th>
<th>Next iteration</th>
<th>Total cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>42</td>
<td>8</td>
<td>50</td>
</tr>
<tr>
<td>Cache (initial hit)</td>
<td>8</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>SPM (filled each time)</td>
<td>34</td>
<td>8</td>
<td>42</td>
</tr>
<tr>
<td>SPM (optimised)</td>
<td>8</td>
<td>8</td>
<td>16</td>
</tr>
</tbody>
</table>

However, this table belies two of the problems with this system. While the overhead of using the SPC and SPM does not appear to be slower than an empty cache, in doing so we need to run three extra instructions (the *smi* and two *jumps*). Thus the overall size of the program becomes larger. For a loop of size 8, iterated twice, these three new instructions represent almost a 19% increase in the number of instructions executed, which will clearly alter the overall runtime. As seen in the previous section, the greater the number of iterations in a loop, the less impact these instruction additions have on performance. In the optimised case, where only the two extra *jumps* are called each time the loop is executed, and the Controller runs once over the life of the program for each loop copied into it, much of this overhead is removed.

Another overhead problem is encountered when an entire loop is copied into the SPM, but only a fraction of that loop is executed (due to the presence of *if* statements inside the loop). For a loop even as small as 10 instructions, where only 6 are run (first 4, last 2), then the initial fill cost of the cache is 36 cycles, compared to 37 for the SPM. However, this is only a problem if the extra instructions copied are never run during the iterations of that loop. If they are run in subsequent iterations, then there is no loss in copying them early.

It has been mentioned a few times that in many cases the optimised *smi* placement is more beneficial than the original placement proposed. In fact, the optimised placement provided the only two test results where the SPM performance out did that of the cache. The advantage of the optimised placement is primarily in that it saves all of the overhead due to repeated SPC operation and loading the SPM. The SPM is loaded once with all the instructions needed at the start of execution, and then only jumped into when a loop is run from it. This cuts back on execution time and power consumption (although it means the Controller is poorly utilised). It is also valuable in avoiding the loop in loop problem. However, since the Controller is only called at the start of the program (generally called in the programs *main* function), there is a limit of 1024 instructions being placed in the SPM for the life of the program. For a program like cjpeg, with > 15000 instructions, it would not be possible to copy enough loops or instructions to prove beneficial. On top of this, there is still the extra 2 instructions overhead for every loop called.

Regardless of which *smi* placement scheme is used though, a major limitation to this experiment is the type of instructions that can be copied into the SPM. For this experiment, syscalls and library functions were copied into the simulator memory during initialisation (ie stored in boot loader). This means that the
simulator was not dependent on the local machine or OS, but solely on the information stored in its memory (as would occur in a real embedded system). However, these library files were not available in C or Assembler format. Therefore it was not possible to alter and recompile these files with additional \textit{smi} instructions inserted. As seen in programs like rawcaudio, most of the discrepancy between cache and spm performance can be attributed to these library functions.

A previously unmentioned limitation with the experiment is that \textit{smi} insertion and address maintenance is done by hand. Often experimental results can be tweaked by hand to look better, however, in this case, the shear number of instructions that need to be altered in order to come close to 100\% utilisation of the SPM is too large to do by hand. In this experiment, the largest number of \textit{smi}’s inserted was 13 for cjpeg, with only 1 inserted for both adpcm benchmarks. Actually adding the instructions is time consuming, but not too difficult. The main difficulty is keeping track of the locations of these instructions in the SPM, and updating the program jump locations to accommodate this. The \textit{jump} location of any loops inside the block being copied into the SPM also have to be updated with their new addresses.

The program style also seems to be important. Programs with a small number of very large loops seem to perform well with the SPM, as long as these loops account for the majority of execution time. Programs that jump around a lot and cause a high degree of cache thrashing also perform comparatively well with a Scratchpad Memory, providing there is a substantial number of loops that can be copied into the SPM.
FURTHER ANALYSIS

It is not all bad news though, for there is one saving grace to the results produced. They were taken and given in clock cycles, in order to compare raw performance. However, one of the advantages of a SPM versus a cache, is that there is no tag checking (or other bits), and so access times are faster. In fact, the access time for a 2 Kb cache is 1.3 ns, compared to 1.05 ns for a 8 Kb SPM [Janapsatya et al, 04].

If we assume that the DRAM used is 10 ns (and still with burst mode), then the clock can be set to 1.05 ns for the SPM configuration, and 1.3 ns for the cache configuration (the memory used is actually SDRAM, so it can handle a change in clock cycle), producing the following results:

The results now show that the SPM configuration is much more comparable to using a cache. The optimised smi placement now out performs the cache in four benchmarks, and in the overall average as well. In fact, despite all of the limitations mentioned previously, the standard smi placement technique almost matches cache speedup.

While it is a little cheeky to show performance improvements by simply increasing the clock frequency (roughly from 760 MHz to 950 MHz), realistically, this is one of the prime advantages provided by using a SPM. However, the large array of limitations previously mentioned still exist here, and there is no indication of how much more energy might be used by increasing the clock speed.
There are three key areas that need addressing in order to try and fully utilise the SPM configuration proposed by Janapsatya. These are the same problems previously identified as limitations.

It would be beneficial to have a software compiler to automatically insert \textit{smi} instructions, and keep track of their corresponding addresses in both memories. This compiler would also need to update any \textit{jump} addresses in the code as well, to ensure the proper use of the SPM, and maintain the Basic Block Table in the Controller. With this tool it would be possible to insert $> 20$ \textit{smi}'s in each program. Janapsatya estimated that an average of 70 \textit{smi} instructions needed to be inserted over the tested benchmarks to see a dramatic improvement over the cache configuration (as much as 120 for pegwitenc).

A better procedure to identify those loops and other areas of code (basic blocks) that need to be copied into the SPM would also help refine performance improvements. This procedure should also be able to identify basic blocks of code that are indiscriminately linked and should be copied simultaneously (as outlined in the introduction), rather than simply copying loops only. Ideally this procedure would also take into account the size of a block, how many times it was iterated, and the cost of running the Controller, in identifying those blocks to copy into the SPM.

Furthermore, it may even be possible to utilise the SPM like a giant block cache. Large blocks of code could be copied into the SPM and run from there. When this block has completed execution, it would be replaced with the next block of program code. This would be particularly beneficial for small programs, or programs with a large degree of temporal locality.

The third key area to address is optimising library functions by copying them into the SPM as well. This should be handled by the compiler though, so there is not much to comment on here.

On another note, one possible solution to the overhead problem is a modified version of the optimised \textit{smi} placement. As observed, filling the SPM once at the start dramatically reduces the overhead caused by the SPC, but reduces the overall number of instructions that can be run from the SPM. However, if an extra SPM register was added to the hardware, it would be possible to keep track of what blocks of code were stored in the SPM. Through clever use of the SPM and the register, it would be possible to store $> 1024$ instructions in the SPM with minimal calls to the SPC. Obviously the compiler would need to keep track of this … a costly problem. In addition, at least one extra instruction would be run each time a loop was encountered (ie check the register first). However, it is estimated that the impact of this extra instruction would be minimal compared to having to copy instructions into the SPM each time it was used. It is unknown though what impact this would have on energy consumption.

While size and execution time are important, in today’s mobile world, power is considered by most as the key constraint in embedded systems [Marwedel, 2002]. Thus the immediate next step is to calculate the system’s energy consumption. Energy profiling of the hardware model will provide details to accurately estimate the energy usage of this system. It is necessary then to determine if the SPM configuration is more energy conscious at this stage than that of the cache configuration. If it is, then further research and optimisations are warranted and welcomed.

Another immediate concern is to look at the footprint of the SPM and the Scratchpad Memory Controller. If the size of the Controller and extra control wiring is dramatically larger than a cache, then it seems futile to pursue research in this area.
The results of this experiment found that the use of a Scratchpad Memory and corresponding hardware Controller were neither promising, nor comparable to that of a standard cache. Over the nine Mediabench testbenches used, the cache system produced an average speedup of 41.7% over using main memory alone. Using a Scratchpad Memory only produced a speedup of 25.1%, which was raised to 30.4% with an alteration to the code modification process.

Even when absolute runtime was observed, as opposed to the number of clock cycles until completion, the cache still outperformed the Scratchpad (27.9% vs 24.8% speedup). Only in the optimised smi placement case was the Scratchpad faster (30.2% speedup).

There is still much room for improvement and research in the area, particularly in the testing of an automated compiler and a more effective method of block detection. However a serious look at energy consumption and physical hardware footprint are needed before continuing down this path.
REFERENCES


Marwedel, Peter, “Embedded Software: How to make it efficient”, DSD 2002 conference, Dortmund, Germany


Special thanks to Seng Lin Shee, Andhi Janapsatya and Jorgen Peddersen for their advice, patience, and programs
AUTHOR’S COMMENTS

There are a number of aspects to the experiments that I undertook which may seem unusual or questionable. Admittedly some of these ‘quirks’ did not become apparent until after the project was completed, and there was no time left to redesign the experiment. As such I felt it prudent to write this appendix to shed some light on some of the decisions made.

Cache size
The cache size used was 2 Kb, compared to an 8 Kb Scratchpad. Research by Andhi found that an 8 Kb SPM was the ideal size for a performance speedup, and thus I chose to implement an 8 Kb SPM. The main reason for the small cache however, was that this was the cache size used in the hardware model. In order to verify the accuracy of the software simulator against the hardware model I left the cache as 2 Kb. Furthermore, the cache was direct mapped, and only had a line size of one instruction. While these may not be realistic for large scale GPP, a small cache is acceptable in cheaper, smaller embedded systems. In fact, some embedded systems don’t even have a cache at all.

Burst mode memory
The memory system used has received quite some criticism, and rightly so. The burst mode employed is not true burst mode. It does not load an entire line into the cache in one go. Rather it takes a long time to load the first word (10 cycles), and then returns subsequent words very quickly if they are in the same line (1 cycle). These subsequent words are only returned when requested, i.e. there is no preloading. The rationale again was that I was trying to match the hardware model for comparison purposes. In defence though, in small embedded systems, the bus sizes are often only large enough for one word at a time anyway.

The configuration of the instruction memory system would appear to bias the SPM over the cache system. However, since the results found that the SPM did not perform as well as the cache anyway, this only further confirms the non promising results obtained so far.

Block copying
In the experiment I only copied loops, rather than large blocks of code that might be beneficial. At the time the rationale was that the only profiler available was one that identified loops. The graph partitioning procedure described by Andhi in his paper was not available for use in this experiment.

Testbench choice
Without being able to copy library functions, perhaps using Mediabench testbenches was not the best choice in trying to show the advantage of using a Scratchpad. Again, initially these benchmarks were used to emulate the results obtained with the hardware model, and to try and confirm the estimative results obtained by Andhi’s research. It was not anticipated at the start of the project that library functions would have had such a large impact on the results.

However, despite these choices, I feel that I have learnt quite a lot during this project. I have definitely come to realise that research and design is a lot tougher and more involved than I had previously thought. I was simply unaware at the start of just how many different aspects and little details needed to be considered and examined. The experience of sitting down with a large, established piece of code, and then trying to understand and modify it, was quite valuable. So too was being exposed to the problems of trying to use other peoples’ code, particularly when I was not privy to how it was written, or its intended method of usage.

I believe that this project has been invaluable in my learning experience as a student, a potential researcher and a future designer. I am now more aware at how large these tasks are, and how complicated they can quickly become. More importantly, I now have experience with the design and research process, and have new tools and methods to utilise next time … let alone a better knowledge of some things that I need to improve upon.