DYNAMIC CACHE SWITCHING IN RECONFIGURABLE EMBEDDED SYSTEMS

John Shield, Peter Sutton, and Philip Machanick

School of Information Technology and Electrical Engineering
The University of Queensland
Brisbane Australia 4072
email: xue@itee.uq.edu.au, p.sutton@itee.uq.edu.au, philip@itee.uq.edu.au

ABSTRACT

The idea of changing cache attributes to suit an application has been explored for single programs. As the popularity of reconfigurable softcore systems grows and these systems increasingly use operating systems and run multiple applications, the possibility arises of dynamic cache switching to improve performance. This paper presents the new idea of dynamic cache switching in a preemptive soft-core system. Such switching optimises the cache structure on a context switch or transition between applications. A practical solution for implementing cache switching in a reconfigurable softcore system is presented. For the design of the switching algorithm, this paper explores the mismatch of cache optimisations between applications. Focusing on the optimisation only is not enough for evaluating cache switching, as not all the applications are optimised and also the overheads in cache switching may not justify the improvement.

1. INTRODUCTION

In recent work with embedded environments, operating systems are increasingly being used on FPGA-based softcore processors. While they still have a subset of applications that constitute the workload of interest, they must also cater for a multitude of secondary programs that may be run. This paper specifically addresses cache optimisations for the workload of interest running on a softcore CPU, with such optimisations happening at transitions between applications, such as context switches.

We propose a reconfigurable multi-cache architecture for a preemptive operating system, where caches are customised for a subset of applications on the system. The caches are changed on a context switch depending on the overhead of the cache switch and the performance difference when changing between the previous application's cache and the new one.

There are three related categories of previous work that only partially address the design issues involved with dynamic cache switching.

It has been shown that cache reconfiguration during the operation of a program can give substantial improvements in program performance [1-3]. This work has looked at changing the block size, associativity, cache size, cache line size and way prediction during the operation of individual programs. However, this previous work only considers single programs and does not consider how runtime cache optimisation might work under a preemptive operating system. A preemptive operating system can context switch to a task where the optimisations may prove a hindrance. Furthermore, the work presented in this paper looks at the optimisation differences between multiple optimised caches, instead of between a generic cache and an optimised cache.

Some related work in the area of task switching looks into cache-related preemptive delay [4-6]. This work has concentrated on the cost of flushing part or all of a cache when blocks of memory overwritten by a previous task need to be refetched. The focus is on calculating these costs to improve the efficiency of scheduling algorithms. None of this earlier work has explored the effect of cache optimisations in a preemptive system.

A different methodology to avoid the preemption problem is through a cache-aware operating system. This involves the operating system or program explicitly controlling things such as cache partitioning [7-9]. Some or all of the tasks are allocated a sub-portion of the cache so there are no cache conflicts from task switching. This approaches the problem from a partitioning perspective, and does not consider cache optimisations crossing application boundaries.

This paper briefly describes the overall system that is under development. It then focuses on the cache switching algorithm, where the issue of *optimisation mismatch*, the situation when applications are run on the customised caches of other applications, is explored.

2. OVERVIEW OF WORK

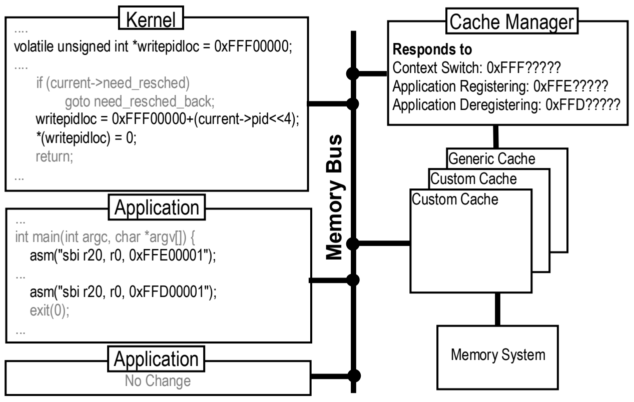
Dynamic cache switching is about changing cache behaviour during runtime to match the application that is currently active. It uses a generic cache controller and a number of specialised cache controllers specific to particular applications. The generic cache controller is used with un-profiled applications or applications that do not load the cache heavily. The specialised pre-built cache controllers are switched in and out of operation in response to profiled applications that are linked to the specialised caches.

Optimisation of a subset of the applications, as used in dynamic cache switching, has not been seriously considered before. Previous work attempted to optimise the entire system, or provide an optimised generic memory system and cater for all the applications. This methodology is a response to the way embedded systems now straddle the boundary between a generic computing system and a custom computing system.

Unlike a generic computer, embedded systems with softcore processors usually have a set of known primary applications that run for the majority of the time. These applications are usually known at synthesis time which means customised caches can be built to cater for them.

There are three main things needed to achieve dynamic cache switching: a cache management system, suitable cache configurations to switch between, and analysis to decide which caches to switch between. These requirements are described below.

2.1. The Cache Management System



```
....
volatile unsigned int *writepidloc = 0xFFF00000;
....
    if (current->need_resched)
        goto need_resched_back;
    writepidloc = 0xFFF00000+(current->pid<<4);
    *(writepidloc) = 0;
    return;
...
```

```
...
int main(int argc, char *argv[]) {
   asm("sbi r20, r0, 0xFFE00001");
...
   asm("sbi r20, r0, 0xFFD00001");
   exit(0);
...
```

Responds to
Context Switch: 0xFFF?????
Application Registering: 0xFFE?????
Application Deregistering: 0xFFD?????

Fig. 1. The Cache Management System

The most important thing about the architecture for implementing dynamic cache switching is that both application and kernel see a single contiguous memory, as with any normal cache. This is important as the designer does not want to adjust every application in their system, and would be unable to if source code is not available.

A cache manager hardware module listens on the memory bus for memory accesses to specific predefined memory locations outside of the main memory address space. Specifically it looks for a memory access from the kernel's scheduler that lists the current process ID and for applications registering or deregistering the current process ID as one that may require a customised cache.

This approach means that applications that are to be optimised require a simple modification to perform a single memory write on startup and a single write when they end. The kernel is modified to perform a single write (specifying a process ID) on program startup or when context switching. Other applications are unchanged.

The cache manager will then be able to trigger off context switches or program start up. It will know whenever an application with a specialized cache is being run, and decide the appropriate action based on a lookup table calculated during compilation. This method also opens up the possibility of multiple caches for different parts of the same application, as the application can reregister itself to a different customised cache. Optimisation of applications without source code is possible if runtime profiling is implemented. However, this is outside the scope of this work. This system has been illustrated in Fig. 1.

2.2. The Caches

As mentioned earlier, previous work has looked at adjusting the block size, associativity, cache size, and way prediction of caches. While the implementation of dynamic cache switching could allow for more exotic cache variations, previous work has found significant performance variation when varying only these standard parameters. This work therefore will only consider variations in these standard cache attributes.

When implemented on a softcore system, dynamic cache switching makes better use of the limited resources available than previous work that attempts to optimise for every case. The resource problem became readily apparent when initial work was started on implementing the hardware of the custom caches. While it may look attractive to have the full range of cache adjustments available, this neglects the fact that memory needs to be put aside for implementing the cache tags of the worst case scenario. Furthermore, greater reconfigurability means more routing and multiplexers that can decrease the maximum clock speed of the softcore processor. This restriction in hardware resources makes implementing only a small number of cache controllers more desirable than allowing for the full range of options.

What this work does share with previous work is the idea of reusing the available memory resources between different caches. A side effect of this is the loss of data from flushing that occurs on a cache switch. It also means that the cache either needs to be write-through or have the overhead of flushing data back to the main memory whenever a cache switch occurs.

2.3. Analysis for Determining Cache Switching

Previous work focused on the improvement in relation to a generic cache. This work looks at the optimisation differences between various customised caches as part of switching algorithm.

The performance loss from cache switching comes about from two main things: cache flushing of data that will later need to be refetched, and reconfiguration time of the cache changing to the new settings. This performance loss is shown by CR, CSP and CF in equation (1). Where CR and CF depend on the morphing techniques used during the cache switching and CSP is the time period these losses occur in. The calculations and methodology for finding CR and CF are beyond the scope of this paper.

The possible performance improvement comes from switching the cache to a new configuration when there is a context switch. This is illustrated by (ETP_1-ETP_2) in equation (1). Where ETP_2 is the performance of the application when the cache is not switched, and is ETP_1 the performance of the application when the cache is switched to the cache optimised for the application.

Equation (1) shows the possible improvement that can be gained from cache switching on a single context switch. This is not to be confused with the overall improvement.

$$improvement = (ETP_1 - ETP_2) - \left(\frac{CF + CR}{CSP}\right) \times 100 \qquad (1)$$

ETP_1 = Execution Time % of current cache: Percentage of clock cycles taken by the current cache to execute compared to the control cache.

ETP_2 = Execution Time % of new cache: Percentage of clock cycles taken by the new cache to execute compared to the control cache.

CR = Cache Reconfiguration time: Clock cycles taken by hardware to reconfigure the cache settings.

CF = Cache Flushing overhead: Number of clock cycles used refetching flushed data that otherwise would have stayed in the cache.

CSP = Context Switching Period: Clock cycles from start of one context to the start of a new one.

The simulation tools that were built generate the number of clock cycles that cache traces (from a microblaze softcore) take to run on each cache. This is the sampled ***execution time*** of an application when run on a cache.

The cache traces obtained are of different lengths so the execution time results need to be normalised. A control cache was chosen with theoretically ideal cache settings: 512Kbytes memory, 16 ways, block size of 1, and Pseudo-LRU replacement policy. The execution time for a result is divided by the control cache's execution time. This gives a normalised execution time, an ***execution time percentage*** (**ETP**) relative to the control cache.

The ETP can be thought of as the normalised performance of a cache for a particular application. The ***ETP difference*** (**ΔETP**) is used to show a ***performance*** difference, either between two applications for the same cache, or one application between various caches.

The ***optimisation mismatch*** is used to describe a type of ΔETP that shows how much performance loss there will be when a cache switch occurs and an application is run on a cache tailored for the previous application. This is the ΔETP of an application that is run on the two caches.

This optimisation mismatch tells the switching algorithm when it is worthwhile to switch to a new cache configuration. If the optimisation mismatch is lower than the overheads of cache switching given by CR, CF and CSP, then it is not worthwhile to perform a cache switch.

3. EXPERIMENTAL SETUP

The experimental results concentrate on one aspect of the work, the cache optimisation mismatch. Finding the optimisation mismatch helps answer the question of when caches need to be switched.

Testbenches were run on a Xilinx ML401 board, running uClinux [10] on a MicroBlaze softcore processor [11]. A custom data gathering system was created on a separate board to record memory bus traces. The relevant data was stripped of its memory access delays and then was placed in a cache simulator which ran through the traces adding the memory timing for different caches.

The analysis in this paper focuses on data memory accesses. As the benchmarks have relatively small program sizes, there are not many instruction cache misses.

For these experiments execution time is the main consideration. However, the methodology can easily be converted to evaluate energy usage as each memory component is modelled in the simulator for the timing.

Variations of memory size, number of ways, block size, and replacement policy of the cache are considered in this experiment. The experiment used 80 samples of recorded memory access patterns to find the mean ETP for each application under various caches. To ensure that sampling variance is not overpowering the results, the margin of error for each mean ETP is calculated for a 95% degree of confidence and compared with the significance of the data.

The mean ETP can then be used as a measure of the performance of applications run under various caches. For all the results a mean ETP is used although not explicitly mentioned.

4. RESULTS

A number of benchmarks from the Mibench test suite [12] were ported to MicroBlaze. Automotive, network, security and telecomm were chosen as likely subsections of the test suite to be used in an embedded system. Analysis was performed in three parts, allowing for narrowing of the search space each time.

First, the amount of the possible optimisation mismatch was found for the search space. This showed which applications may need a custom cache. Second, the optimisation mismatch between applications was mapped over the search space. This shows trends in how applications differ in their reaction to changes in the cache. Finally, the optimisation mismatch that will occur with cache switching is shown.

4.1. Optimisation Mismatch Possible for Applications

Not all applications benefit significantly from cache optimisation. To find out which applications can benefit, the ETP of their various cache configurations were then compared with their optimal solution to see if there was any difference. This is comparing the ETP of an application between caches.

As seen in Table 1, the Djisktra, Blowfish and Rijndael testbenches had significant ETP performance changes over the cache variations chosen, so this paper will concentrate on the results for these applications.

The other testbenches failed to provide significant statistical difference in their performance when the cache attributes were varied. That only some of the applications show significant responses to cache variation fits with the theory that optimising everything is not necessary.

Table 1. Largest ETP Difference from Optimal in Testbenches

Test Bench	ΔETP from optimal	Test Bench	ΔETP from optimal
Djisktra	40.876%	ADPCM	4.139%
Blowfish	18.343%	Bitcnts	2.490%
Rijndael	14.603%	CRC	0.353%
Basicmath	0.028%	FFT	0.248%
SHA	0.217%	GSM	3.655%

4.2. Comparison Between Applications

The ETP of each cache was compared between applications. This gives an idea of ***cache trends*** between applications, increasing or decreasing a particular attribute will either increase or decrease the difference in performance between the applications.

Finding the exact optimisation mismatch between applications is an extra dimension when searching the design space. This mismatch is therefore analysed only after the design space has been narrowed.

Comparisons are shown between the testbenches Blowfish and Djisktra, and Rijndael and Blowfish. The Rijndael and Djisktra comparison was sufficiently similar to the results of the Blowfish and Djisktra testbenches that the analysis would be repetitive. Other comparisons did not have enough significance in the ETP difference (ΔETP). The replacement policies used were Pseudo-LRU and random, however these generated only a maximum of 0.5% difference so these comparisons are not shown here.

The graphs in this section show the ΔETP between two applications for the same cache. Each data point represents a cache with various settings and shows the difference in their performance from optimal (ΔETP) between the two applications for that cache.

The data points for all the possible cache configurations are shown in the graphs. They are arranged along the X axis by only two of their attributes; the attribute labelled in the axis and the attribute in the legend. Multiple data points for each X are due to variations in the other attributes.

4.2.1. Comparison Between Blowfish and Djisktra

While Blowfish and Djisktra showed similar attributes when considered individually, when the results were overlaid there was a difference in how they behaved.

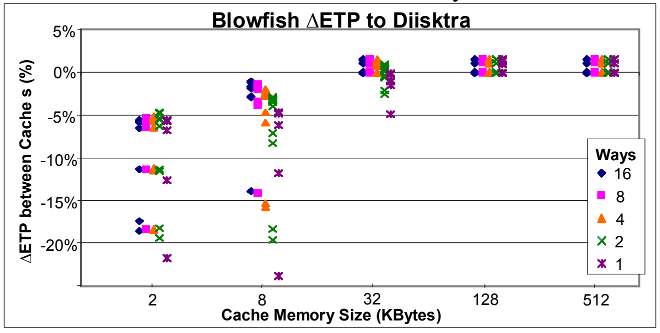


Fig. 2. Blowfish/Djisktra comparison of Ways and Cache Memory

Fig. 2 graphs the variation of ETP for Blowfish compared to Djisktra. Each data point corresponds to a cache, with varying attributes, that was simulated. The data points are graphed by memory size along the X axis and further clustered by number of ways around each memory size location. The Y axis maps the difference of the first mentioned application Blowfish, to the second application Djisktra. Djisktra showed less tolerance for small cache sizes than Blowfish. Also a higher number of ways improved Djisktra around 8KB of memory, while playing no significant role when Blowfish is considered.

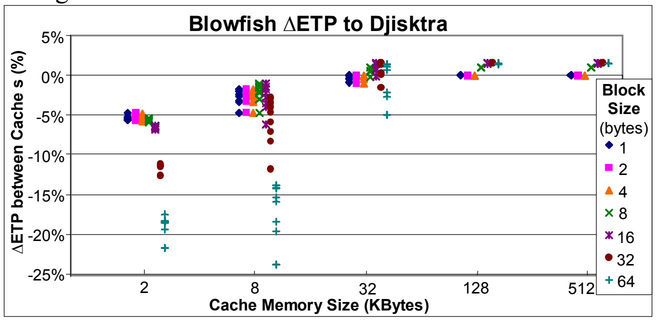


Fig. 3. Blowfish/Djisktra comparison of Block Size and Cache Memory

In Fig. 3 the data points are grouped by memory and block size. Blowfish compared to Djisktra handled larger block sizes better when the memory was low, while Djisktra had a performance improvement with larger block sizes when the memory sizes were larger.

The data shows that when designing a cache for one application, attributes that are not important for that case can play a significant role in how another application responds. Looking at the data for each application individually (not shown here), Djisktra overshadows Blowfish in its sensitivity to the cache attributes such that Blowfish would work reasonably well under a cache that was created for Djisktra. It highlights that sometimes switching from a cache optimised for one application to a cache that is optimised to another may not be necessary.

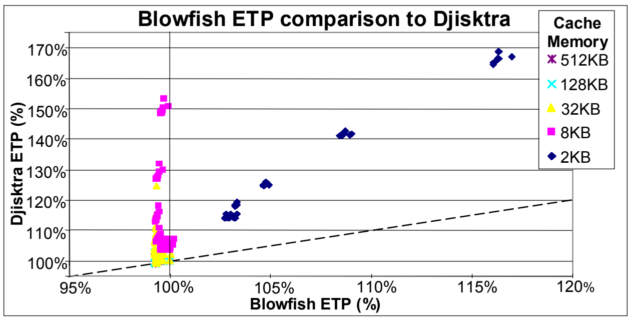


Fig. 4. Blowfish/Djisktra comparison of Cache Memory and ETP

Fig. 4 remaps the results so the ETP of Blowfish can be directly compared with Djisktra. This shows whether the revealed trends with the performances have some correlation between applications or are independent. The previous graphs only showed the difference between the applications.

The dotted line is a line of exact correlation between the two programs. Deviation from the dotted line means that there is a cache trend that is stronger for one application than the other. In this case the cache trends are all stronger for Djisktra, reflecting how Djisktra overshadows Blowfish in the cache trends.

The vertical "line" of points in Fig. 4 shows that at cache memory size 8KB (and to a lesser degree 32KB), Djisktra exhibits significant differences in performance while this trend is not present with Blowfish. This confirms the earlier conclusion that certain cache settings can impact one application but play no role in another.

For a cache memory size of 2KB, the diagonal "line" of data points shows how Blowfish and Djisktra trends correspond to a degree. When the trends correspond it means that the optimal cache for Blowfish and Djisktra may overlap for those settings.

4.2.2. Comparison between Rijndael and Blowfish

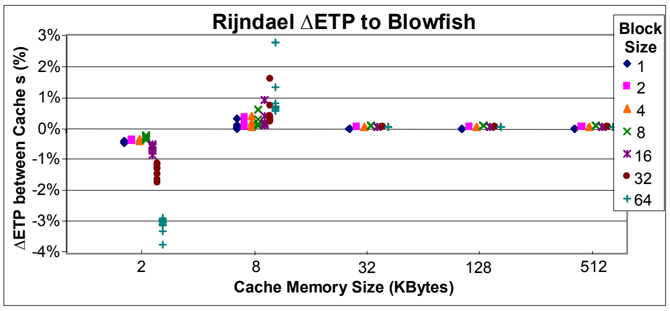


Fig. 5. Rijndael/Blowfish comparison Block Size and Memory Size

In Fig. 5 increased block size gives an improvement for Rijndael over Blowfish at 2KB of cache memory. While this trend reverses at 8KB of cache with block size giving an improvement for Blowfish over Rijndael instead.

Looking at the dotted line in Fig. 6 and considering the cache trends, the cache trends are stronger for the Rijndael at 2KB while they are stronger in the Blowfish at 8KB.

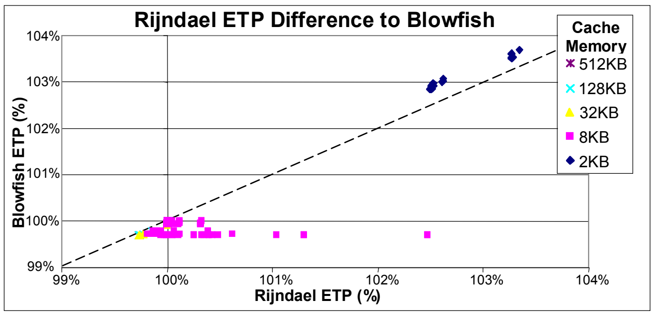


Fig. 6. Rijndael/Blowfish comparison of Cache Memory and ETP

This difference in which application has the stronger behaviour is important, as it shows that one application does not overshadow the other all the time. This differing behaviour suggests that there is optimisation mismatch that is beneficial for cache switching. Unfortunately this difference is not relevant in the next section as the assumption is made that cache memory size is fixed.

4.3. Valid Optimisation Differences

The graphs of the ΔETP in the previous sections show differences in the cache trends and allow for the different dimensions in cache configuration to be explored quickly. They do not show whether the configuration points would actually be used in a real situation. ***Optimisation mismatch***, ΔETP between the current cache configuration and its optimal cache configuration can be found if the cache is configured for another application. It can also be found when a cache that tries to optimise between multiple applications is used.

In a reconfigurable embedded system the available fast memory that can be used by the cache is a limiting factor. Cache sizes are therefore considered a fixed factor in this analysis and only 8KB and 2KB cache sizes were studied.

Table 2. Worst Case Optimisation Mismatch

Djisktra		Blowfish		Rijndael	
Cache Size 8KB	Cache Size 2KB	Cache Size 8KB	Cache Size 2KB	Cache Size 8KB	Cache Size 2KB
22.676%	31.389%	0.919%	14.253%	4.048%	11.290%

Table 2 shows the worst case mismatch that can occur. This is how much the ETP can stray from its optimal (lowest) value when varying cache attributes other than the memory size. This is useful for giving perspective to mismatches discussed later. Looking at the percentages of mismatch shows how great an effect the mismatch will have on the application performance. Comparing it to the maximum mismatch shows the magnitude of the optimisation conflict between the two applications.

Table 3 shows mismatch when the cache is optimised for only one of the applications. This is the most relevant situation when cache switching occurs. For the majority of cases when one of the programs is optimised, the cache works reasonably well for the other applications.

Table 3. Mismatch when Optimised for One Application

	Optimised for Djisktra		**Optimised for Blowfish**		**Optimised for Rijndael**	
	8KB 16 Way 16 Blocks	2KB 2 Way 1 Blocks	8KB 1 Way 64 Blocks	2KB 16 Way 4 Blocks	8KB 16 Way 16 Blocks	2KB 16 Ways 4 Blocks
Djisktra	NA	NA	12.771%	0.799%	0.000%	0.799%
Blowfish	0.032%	0.148%	NA	NA	0.032%	0.000%
Rijndael	0.000%	0.038%	0.436%	0.000%	NA	NA

However, the optimal Blowfish configuration at 8KB memory size turned out to be a bad configuration for Djisktra, resulting in a 12.8% execution time increase from the optimal Djisktra solution. This shows a significant conflict of cache trends as the worst possible setting shown in Table 2 is 22.7%.

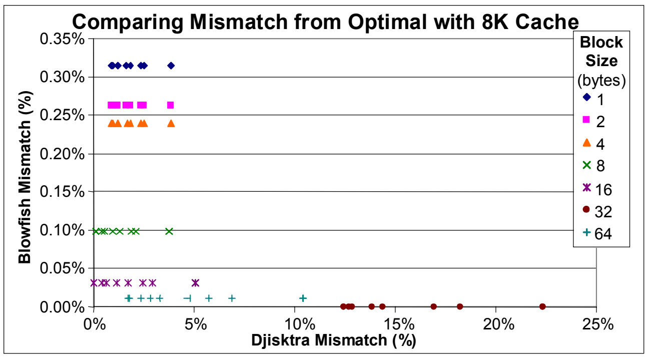


Fig. 7. Mismatch with Djisktra and Blowfish at 8KB Cache Size

In Fig. 7 the mismatched optimisation between Djisktra and Blowfish testbenches can be seen for various block sizes. Blowfish runs better with a large block size while Djisktra works better with smaller blocks.

The high optimisation possible with these few testbenches shows that dynamic cache switching is more useful when there are a number of applications that are to be optimised. The greater variance found with more applications will make it less likely that one application will have a much higher magnitude of optimisation mismatch compared with the others and more likely that there will be significant mismatches between applications.

5. CONTRIBUTIONS AND FUTURE WORK

This paper contributes three main things.

First, it has explained how dynamic cache switching is particularly suitable for a preemptive softcore system. There are tangible performance benefits gained from having custom caches, but a framework that optimises for the entire system would be unsuitable. By contrast, previous work covers global approaches that optimise everything in either a generic or very specific way.

Second, it shows a novel and practical way to implement cache switching with minimal changes required in software.

Finally, it explores the concept of cache mismatch, an improved way to measure performance improvement in dynamic cache switching. Previous work measures improvement in relation to a generic cache.

In future work, the effect of the hardware overheads involved in cache switching will be considered. This combined with the research into optimisation mismatch will be used to generate the cache switching algorithm.

6. REFERENCES

[1] Gordon-Ross, A., Vahid, F., Dutt, N., "Automatic tuning of two-level caches to embedded applications," in *Proc. Design, Automation and Test in Europe Conference and Exhibition,* , vol. 1, pp. 208-213, 16-20 Feb. 2004.

[2] Chuanjun Zhang, Vahid, F., Lysecky, R., "A self-tuning cache architecture for embedded systems," in *Proc. Design, Automation and Test in Europe Conference and Exhibition*, vol. 1, pp. 142- 147, 16-20 Feb. 2004.

[3] Hu, J. S., Kandemir, M., Vijaykrishnan, N., Irwin, M. J., Saputra, H., and Zhang, W., "Compiler-directed cache polymorphism," *SIGPLAN Not.* Vol. 37,, pp. 165-174. 2002.

[4] Stärner, J. and Asplund, L., "Measuring the cache interference cost in preemptive real-time systems," in *Proc. ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools For Embedded Systems*, pp. 146-154, 11-13 Jun. 2004.

[5] Chang-Gun Lee, Kwangpo Lee, Joosun Hahn, etc. "Bounding cache-related preemption delay for real-time systems," *IEEE Trans. Software Engineering*, vol. 27, no. 9, pp. 805-826, Sep 2001.

[6] Staschulat, J., Schliecker, S., Ernst, R., "Scheduling analysis of real-time systems with precise modeling of cache related preemption delay," in *Proc. Euromicro 17th Real-Time Systems*, pp. 41- 48, 6-8 July 2005.

[7] Y. Li and W. Wolf, "Hardware/software co-synthesis with memory hierarchies," *IEEE Trans. Computer-Aided Design*, vol. 18, pp. 1405–1417, Oct. 1999.

[8] Liedtke, J., Hartig, H., Hohmuth, M., "OS-controlled cache predictability for real-time systems," in *Proc. IEEE 3rd IEEE Real-Time Technology and Applications Symposium*, pp. 213-224, 9-11 Jun 1997.

[9] Mueller, F., "Compiler support for software-based cache partitioning," in *Proc. ACM SIGPLAN Workshop on Languages, Compilers, & Amp; Tools For Real-Time Systems*, pp. 125-133. 1995.

[10] Xilinx, "MicroBlaze Processor Reference Guide", UG081 (v6.0), June 2006.

[11] uClinux, http://www.uclinux.org/, January 2007.

[12] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge and R. B. Brown, "MiBench: A Free, Commercially Representative Embedded Benchmark Suite," in *Proc. IEEE 4th IEEE International Workshop on Workload Characterization*, Dec. 2001.