Instruction cache tuning for RTOS-driven multitasking applications

November 4, 2009

Abstract

With the advent of mobile and handheld devices, power consumption in embedded systems has become a key design issue. Recently, it has been shown that cache requirements of the applications vary widely and a significant amount of energy can be saved by tuning the cache parameters according to the needs of the application. To this end, techniques have been proposed to tune the cache for single-task based systems but no work has been done to extend these techniques to multitasking applications. In this research work, we present novel, lightweight and fast techniques for energy-sensitive tuning of the instruction cache hierarchy for multitasking applications. Cache tuning for RTOS-driven multitasking applications is achieved by intelligently separating the user tasks and RTOS components and profiling them in isolation to identify the nature of loops in them. We then apply the proposed techniques to tune a predictor based filter cache hierarchy for instructions for both single-task based applications and RTOS-driven multitasking applications. The proposed techniques are able to identify optimal or near-optimal filter and L1 cache sizes for all the applications tested and are up to an order of magnitude faster than exhaustive cache hierarchy simulation techniques. The proposed techniques are also highly scalable and can be relied upon to predict the instruction cache hit rate for any range of instruction cache sizes after a one-time simulation and profiling.

1 Introduction

Loop profiling enabled cache tuning has been shown to achieve considerable energy savings for standalone embedded applications. While such loop profiling techniques work well for stand-alone applications, there has been no work done to extend them to multitasking systems running on a real-time operating system. The challenge in applying loop profiling algorithms to tuning multitasking systems lies in the inherent pseudo-random nature of control flow in multitasking systems. In a stand-alone application or single-task-based application, there is no sharing of system resources like processor, memory, I/O, etc. In such systems, the functions that need to be performed by the stand-alone application proceed without interruptions because it is the sole master of all resources available to it. However, in a multi-task based system, tasks compete with each other for system resources and end-up pre-empting each other as a result of mutual competition. This pre-emption makes the control flow of

multitasking systems pseudo-random and prohibits direct extension of loop profiling techniques developed for stand-alone applications to include multitasking systems.

With the increasing complexity of modern day embedded applications, RTOS based multitasking systems are fast becoming the norm. There exists a need to extend existing cache tuning techniques developed for standalone embedded applications to multitasking systems. In this paper, we present a framework for energy centric-tuning of the instruction cache for embedded multitasking applications. Our framework is built upon an analysis of control flow of multitasking system and how the pseudo-random flow of control affects execution of loops and consequently, the size of the optimal cache for the multitasking application. We validate our proposed techniques by using it to tune a predictor based filter cache hierarchy for instructions. For all the multitasking programs tested, our techniques are able to successfully predict configurations that are optimal or near-optimal. The proposed methods are also able to achieve speed-ups of up to an order of magnitude compared to exhaustive design space exploration techniques.

The rest of the paper is organized as follows. We discuss tunable cache memories and techniques for tuning the cache parameters in section 2. The cache tuning heuristics that we use in this work for the predictor-based filter cache hierarchy are presented in section 3. In section 4, we analyze the nature of control flow in multitasking application and comment on its determinism. We show that the control flow in multitasking systems is part deterministic and part non-deterministic resulting in a pseudo-random flow. We discuss the impact of this pseudo-random control flow on cache accesses in section 5. We show that the pseudo-random control flow results in two kinds of misses - misses due to instructions of the same tasks or intrinsic interference and misses due to competition between different tasks to occupy the instruction cache or extrinsic interference. We show that misses due to intrinsic interference are much lesser as compared to extrinsic interference through empirical experiments and explain the observation theoretically as well. Based on our observations on the relative impacts of intrinsic and extrinsic cache misses, we propose our tuning framework in section 6. We present our experimental results on instruction cache tuning for multitasking systems in section 7. We present a method to estimate the number of extrinsic instruction cache misses in section 8 and conclude the paper in section 9.

2 Application-specific tuning of cache memories

Cache requirements of applications vary widely [41]. Some may need a larger cache for the best performance while others may need to save on energy and it may be reasonable to use a smaller cache for such applications. When it comes to block sizes too, the amount of spatial locality may be different in different applications and using a one-size-fits-all approach may not be ideal. Tuning cache parameters in an application specific manner has been shown to aid in considerable savings in energy and improvement in performance [29][40]. With these requirements in mind, configurable or tunable cache memories were proposed to tweak cache parameters in an application specific manner. In this section, we discuss tunable cache memories and also present techniques proposed

in the literature for application-specific cache tuning.

2.1 Tunable cache memories

The concept of configurable or tunable cache memories has been prevalent in both softcore and hardcore processors. In softcore processors like Microblaze and Nios II [3][38][27], the cache configuration is chosen at systhesis-time. On the other hand, in hard-core processors [2][18][41] the cache is configured at run-time. For soft-core processors, the user usually chooses a cache configuration before systhesis and generates a customized cache. In hard-core processors, the desired cache configuration can be obtained by initializing registers on the chip that control muxtiplexers and the cache controllers using software code at run-time.

Techniques have also been proposed in the literature wherein the cache adjusts its parameters automatically to the executing application. This is achieved by observing special counters at run-time at regular intervals and efficiently exploring the configuration space to select the best cache configuration [40][12]. Such an approach is known as the self-tuning approach.

While self-tuning is a neat method for cache tuning, the infrastructure for the special counters required in this method may not be always available. Even if the infrastructure is available, the overheads associated the control circuit may defeat the purpose of cache tuning. These problems are addressed by one-shot software-reconfigurable tuning wherein optimal cache parameters for the applications are determined statically and instructions are added to the application to tune the cache at run-time.

2.2 Design space exploration for cache tuning

Cache tuning is usually achieved by obtaining cache hit rates for a variety of cache configurations and then identifying the optimal cache configuration using user-defined tuning heuristics as shown in figure 1. The first step in this process is to obtain the hit/miss rates for a variety of cache configurations. In the next step, the multitude of cache configurations for which the hit rate has been measured using cache memory simulation or estimated using profiling is fed to a cache energy-delay model database like CACTI [36]. CACTI is able to estimate the energy per cache access and the delay per cache access for the input cache configurations. The hit rates, per access energy and delay values are then used to identify an optimal cache configuration using a set of user defined heuristics. A common heuristic searches for a cache configuration that results in the lowest energy consumption [12][31][41].

While the heuristics for selection of an optimal cache is based on system constraints and is left for the system designer to specify, there are many techniques to measure or estimate cache hit/miss rates for multiple cache configurations. The techniques for measuring the hit rates rely primarily on cache simulation of the desired cache configuration by using the program trace as an input for simulation. On the other hand, techniques to estimate the cache hit rates are based on analyzing the locality in the programs either with the help of a compiler or through a profiler. The profiler is fed with runtime information about the execution of the program and identifies potential zones of locality of reference in the application. In this sub-section, we discuss the various options

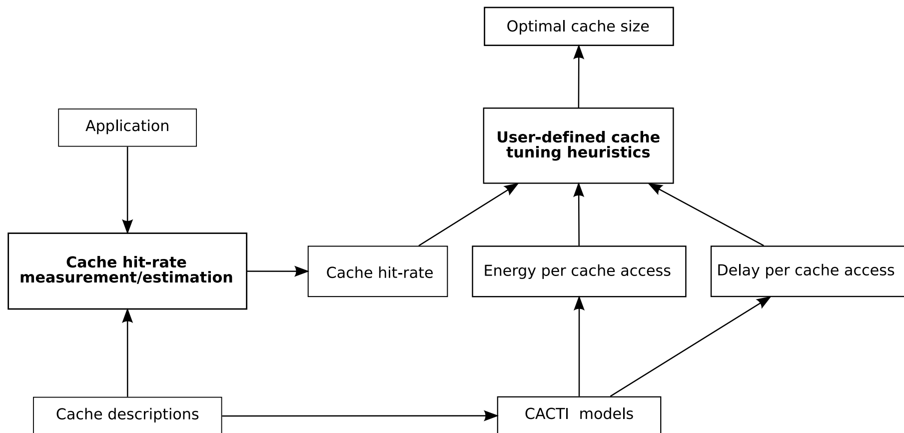


Figure 1: Analytical cache tuning techniques for the filter cache hierarchy

available for design space exploration to identify optimal cache parameters for an application.

2.2.1 Design space exploration using exhaustive cache simulation

In exhaustive techniques for cache tuning, simulation based search methods are used to find optimal cache configurations. In such techniques, multiple cache configurations are simulated to identify the optimal cache configuration. The number of combinations that need to be searched in this approach is very large. This entails simulating the application multiple times and can be prohibitively time consuming even for small applications. Such brute-force search techniques are ineffective in projects with time-to-market pressures.

Therefore, a significant amount of work has focussed on reducing the number of cache configurations that need to be examined to arrive at an optimal configuration. Several tools do exist for assisting designers in tuning a single level of cache. Platune[11] is a framework for tuning configurable system-on-a-chip (SOC) platforms. Platune prunes the search space by isolating interdependent parameters from independent parameters. However, cache parameters like cache size, block size, associativity,etc being inter-dependent, are explored exhaustively which makes this approach prohibitively time consuming.

Heuristic-based methods were proposed to prune the search space of the configurable cache. A genetic algorithm was proposed in [22] to improve upon the exhaustive search used in Platune to produce comparable results in less time. Another technique was presented in [] wherein a cache configuration exploration methodology wherein a cache exploration component searches configurations in order of their impact on energy, which reduces the number of combinations that need to be evaluated. Iterative heuristics were proposed in [39] so that near-optimal cache configuration is reached without actually simulating the entire configuration.

2.2.2 Design space exploration using static control flow graphs

Optimizing compilers have long relied on analysis of program locality to detect data reuse and estimate cache misses. The analysis of data dependencies and reference patterns has been used in a variety of loop transformations to improve locality of reference in the program and reduce accesses to the main memory. Researchers have worked on numerous such re-ordering or transformation techniques to improve the locality of reference in applications like loop interchange [9][20], loop fusion/fission [20] and loop tiling [6][8][16].

In [20], a cost model was proposed to find desirable loop organizations by computing both the spatial and temporal reuse of a cache line. The cost model was shown to be highly effective in deriving loop structures that result in the fewest number of accesses to the main memory. A technique to estimate conflict misses by linearizing array references and then evaluating the amount of cache conflict resulting from memory references was presented in [24]. Cache miss equations were presented in [10] to describe the cache performance of applications analytically. In this technique, deciding whether a reference causes a hit or miss for a given iteration of a loop was equivalent to deciding whether it belonged to a polyhedra defined by the cache miss equations.

Design space exploration using control flow graphs has low overheads because it can be done at compile time without needing to run the application. However, it is insensitive to loops whose boundary is known only at runtime and therefore, offers inaccurate measures of locality for applications that contain such loops.

2.2.3 Design space exploration using runtime trace

Design space exploration techniques that have relied on the runtime trace generally focus on extracting features from the trace that quantify the level of locality in the application. The techniques proposed in the literature rely on these extracted features to predict the hit/miss rates for a range of cache configurations. One of the earliest of such techniques was proposed in [7]. In that work, it was proposed that the miss rate of the cache size of C could be modeled by the equation $MR(C) = C^{-\alpha}$. The parameter α could be found based on the available program trace. Based on the empirical model in [7], a fractal model of locality was also proposed [28]. The argument put forth in [28] was that the sequence of memory references represented a random walk through memory. The jumps from one region of the memory to another are often short and occassionaly long. It was shown that such a randon walk is a special case of fractals and therefore, existing work in fractal theory can be relied upon to model the locality of the trace. The miss rate for a cache of size C is modeled as $MR(C) = AC^{\theta}$ in this work. The different values for A and θ can be obtained using existing data. These models were effective in shedding some light on the nature of locality of the program. However, they focussed only on the cache size and were not detailed enough to observe the effect of changing the block size and associativity on the hit rate. Moreover, for the fractal model of locality, while the instruction stream showed characteristics of a random walk necessary for a fractal model, most of the data stream lacked this.

A detailed model for modeling temporal and spatial locality of the reference stream after breaking the reference stream into time granules was proposed in [1]. Apart from considering the cache size, the model also takes into account

the block size, associativity and cold start effects. To estimate the miss rate, this model has four locality parameters. The first parameter is the number of unique references that are made in a time granule. The second parameter is the collision rate of different cache blocks in the cache. The last two parameters are inputs to a two-stage markov model for analysing the effect of block sizes on the collision rate. The collision rate is derived from the number of unique references and the inputs to the markov model using an expression. These input values to the model are measured from the trace of the application. While this technique offered a feasible way to model a variety of parameters affecting the miss rate, the average error rate in miss rate estimation occasionally reached as high as 23 % for certain benchmarks and configurations. Measurement of locality based on reuse distance, which is discussed subsequently, offered a solution to this problem by reducing the maximum error rate to about 5 to 10 %.

The analysis of program locality based on reuse-distance analysis was initially proposed in [19]. In this technique, the number of distinct data accesses are counted between two accesses to the same data element. After obtaining information about the reuse distance for the data elements, the whole program can be represented as histogram describing reuse distance distribution, where each interval showing the portion of memory references which falls into the same reuse distance range. This reuse distance information can then be used to estimate the hit/miss rate for multiple cache sizes. While the reuse-distance analysis offered a feasible technique to explore multiple cache sizes and offered fairly accurate results when compared with other techniques, there were issues with its scalability. The time complexity of a naive stack distance algorithm is in the order of $O(NM)$ where N is the length of the trace and M is the size of the program data. The space complexity of the distance analysis data structure is $O(M)$ in this case. Indeed, this was the complexity reported in early research work in this area [19]. As the size of programs increased, this complexity was found unacceptable, therefore further research was done to deal with this and an m-ary tree built on a vector was used for analysis in [4]. The same work also showed how to use blocked-hashing in a pre-pass. The first tree-based organization of the data access trace using an AVL tree was discussed in [21]. In the same work, techniques were also proposed to compress the trace vector used in [4]. In [26], it was shown that a splay tree has better performance than an AVL tree. Vector based techniques reduce the time complexity to $O(NlogN)$ but the space complexity remained very high at $O(N)$. Tree-based techniques reduced the time complexity to $O(NlogM)$ with space complexity being $O(M)$.

Relying on the program trace to determine a suitable cache configuration was feasible as long as the program traces were small. The ever-increasing complexity of modern-day applications forced researchers to investigate on efficient program trace collections techniques. Some techniques were also reported to reduce the trace generated by the simulated application and/or simulate multiple configurations in one go [14] [25]. To improve the simulation speed, techniques were formulated to find approximate trace [15] or lossless trace reduction [23] [35] [37].

2.2.4 Design space exploration using dynamic control flow graphs

Lately, cache tuning based on loop profiling has been shown to achieve significant speed-up in the tuning process [33][30]. It relies on tuning the cache based

on results from a one-time loop profiling of the weighted or dynamic control flow graph (DCFG) of the application. DFCGs are edge-weighted versions of the static control flow graph. The DCFG of an application can be obtained by instrumenting the application with counters so that the frequency of a branch can be obtained. Alternatively, a weighted control flow graph can also be obtained by obseving the trace of the application and then deducing the frequency of the edges.

Compared to the static control flow graph based design space exploration, the dynamic control flow graph approach is more accurate as it contains runtime information. The runtime information is useful in identifying the frequent code segments like loops whose boundaries are known only at runtime. Identification of frequent code segments is essential to calculating the working set size for the program and then selecting a suitable cache for the application.

Compared to stack distance based techniques, the dynamic control flow graph offers a faster and scalable solution. Building, maintaining and analyzing a dynamic control flow graph is simpler that computing locality based on stack distance probabilities. Firstly, because the DCFG is at the basic block level, it is simpler to obtain. When the edge weights are computed, it is just a simple increment operation. For example, when a basic block is executed, only a counter needs to be incremented to keep track of the execution frequencies. However, for the stack distance method, the stack has to searched and the probabilities updated for every reference made. This is more involved than building the DCFG. On the flip side, due to a higher level of abstraction, the DCFG based technique is not as accurate as stack distance based measures.

There are three steps in using DCFGs to obtain estimates for suitable cache sizes for the application. First the application is simulated in an instruction set simulator or the actual platform if available. After simulation, the DCFG is available for further processing. Loop exploration algorithms are then applied to obtain the frequent code segments in the application. In the final step, the relative frequency of these loops and their size is taken into account in obtaining an estimate for a suitable cache size for the application. The heuristics used for selecting the cache size are usually defined by the designer and usually measures for energy spent per access to the cache are used to identify an optimal cache size.

3 Tuning Heuristics for Predictor Based Filter Cache

To test our profiling techniques for mulitasking cache tuning, we chose the predictor based filter cache hierarchy as a testcase. This is a common solution for low power cache hierarchies in the embedded systems domain. In this section, we give an overview of the predictor based filter cache hierarchy and also discuss energy-centric heuristics for tuning the same.

3.1 Predictor based filter cache hierarchy

The predictor based filter cache hierarchy, shown in figure 2, consists of a filter cache in conjunction with an L1 instruction cache [32][34][30]. The filter cache is a tiny auxiliary instruction cache whose purpose is to hold the instructions

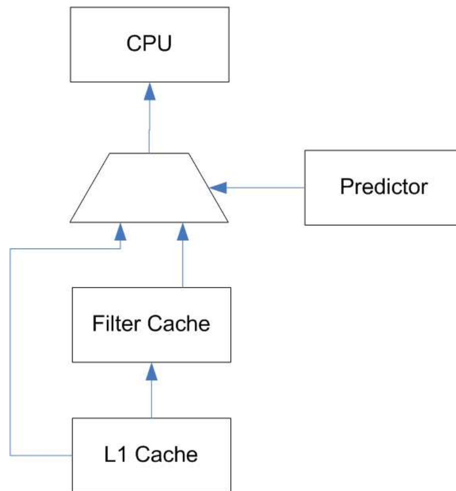


Figure 2: Predictor based filter cache hierarchy

of the many loops inherent in embedded applications. While these tiny loops execute, access to the small filter cache reduces the energy expended in the cache hierarchy. However, the filter cache could be a potential liability for larger loop sizes as they cannot be contained in the small cache.

This is where the predictor becomes useful. Simply put, its role is to minimize accesses to the filter cache when the requisite instruction is not expected to be there. The predictor in a filter cache hierarchy works in a two stage manner. The first level is the cache line level. It assumes that the address of the next instruction to be accessed will be the current value of the program counter plus the instruction size. If the current instruction and the future instruction map to the same cache line, instruction access is directed to the filter cache. If they do not map to the same cache line, a pattern predictor is accessed to decide whether the next access will be from the L1 cache or the filter cache [32]. The prediction process is shown in figure 3.

The important thing to note here is that when an instruction is accessed from the L1 cache, a full filter cache line containing the instruction is moved to the filter cache too. Subsequent access to instructions from the same line are done from the filter cache instead of the L1 cache. So in an ideal case, the predictor can sense that the instruction will be a miss in the instruction cache and direct it to the L1 cache. The proportion of the instructions directed to the L1 cache will therefore be the application wide miss rate for the filter cache.

3.2 Instruction cache hierarchy tuning heuristics

Energy consumption is a prime issue in modern day embedded systems. Therefore, our objective behind cache tuning is to find an ideal instruction cache hierarchy for a given application which reduces the energy consumption of the cache hierarchy without trading off too much performance. Consideration of the performance clause is important because using a smaller cache may help save energy but it will ultimately lead to poorer performance due to the many misses that might occur while accessing it. To meet our tuning objective, we

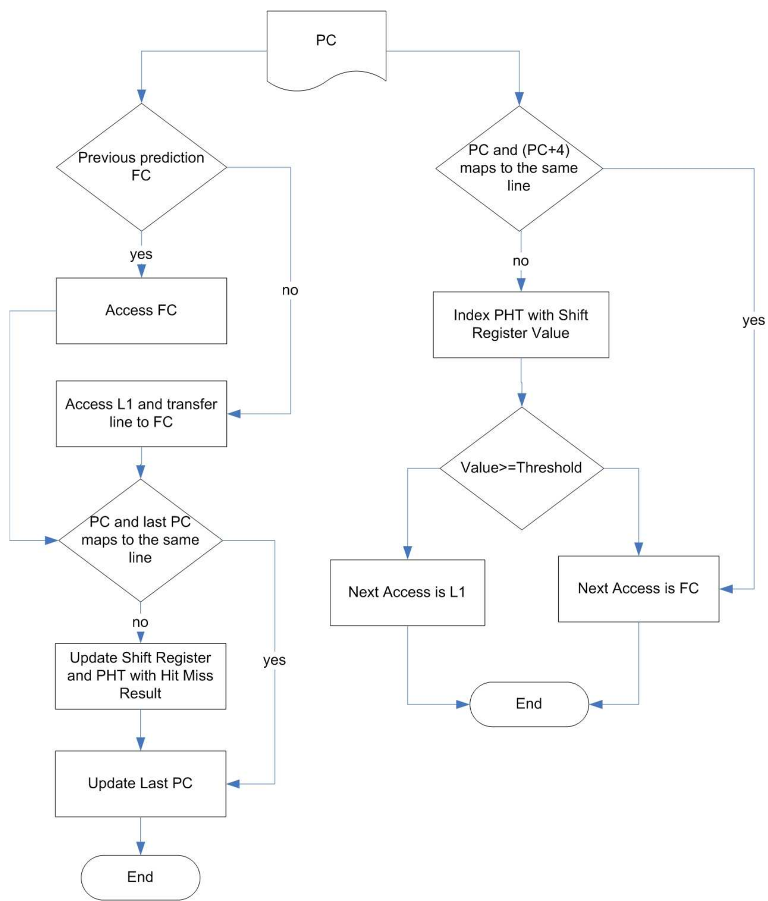


Figure 3: Filter cache prediction methodology

used a yardstick that captures both the energy saved and the performance that is sacrificed to achieve this - the energy-delay product.

For calculating the energy-delay product of the filter cache and the L1 cache in the predictor based instruction cache hierarchy, we look at the following properties of the cache.

- `Cache Access Time`: This is average amount of time spent per cache reference (assuming it to be a hit) to retrieve the requisite instruction from it. In the following equations, this is represented as AT_{FC} for the filter cache and AT_{L1} for the L1 cache.

- `Cache Access Energy`: This is average amount of energy spent per cache reference (assuming it to be a hit) to retrieve the requisite instruction from it. In the following equations, this is represented as AE_{FC} for the filter cache and AE_{L1} for the L1 cache.

- `Cache Miss Rate`: This is the application wide miss rate for a given cache size. In the following equations, it is represented as MR_{FC} for the filter cache and MR_{L1} for the L1 cache.

- `Average Memory Access Time`: An ideal predictor can identify if a reference will be a miss in the filter cache and direct it to the L1 bypassing the filter cache altogether. This way, when the instruction can be a hit in the filter cache, we get the latency of the filter cache and when it will be a miss in the filter cache, we get the latency of the L1 cache. In a predictorless filter cache hierarchy, the latency on a miss would be the access time of the filter cache plus the access time of the L1 cache. The average memory access time for the filter cache and the L1 cache is calculated as shown in equation 1 and 2. In these set of equations, $AMAT_{FC}$ is average memory access time for the filter cache and $AMAT_{L1}$ is the average memory access time for the L1 cache. HR_{FC} is the application-wide hit rate for the filter cache.

$$\begin{aligned} AMAT_{FC} &= HR_{FC} \times AT_{FC} \\ &+ MR^{FC} \times AMAT_{L1} \quad (1) \\ AMAT_{L1} &= AT_{L1} + MR_{L1} \times AT_{L2} \quad (2) \end{aligned}$$

- `Average Memory Access Energy`: The case for average memory access energy calculation in a predictor based filter cache hierarchy is different from that of the average memory access time calculations. When an instruction is accessed from the L1 cache, the line is transferred to the filter cache and subsequent accesses from the same line are made from the filter cache. So, for every access to the L1 cache, the filter cache is also accessed once. The average memory access energy for the filter cache and the L1 cache is calculated as shown in equations 3 and 4. In these set of equations, $AMAE_{FC}$ is average memory access energy for the filter cache and $AMAE_{L1}$ is the average memory access energy for the L1 cache.

$$\begin{aligned} AMAE_{FC} &= AE_{FC} + MR_{FC} \times AMAE_{L1} \quad (3) \\ AMAE_{L1} &= AE_{L1} + MR_{L1} \times AE_{L2} \quad (4) \end{aligned}$$

- `Energy-Delay Product`: The energy-delay product is just the product of the average memory access time and the average memory access energy and is calculated as shown in equations 5 and 6

$$EDP_{FC} = AMAE_{FC} \times AMAT_{FC} \quad (5)$$
$$EDP_{L1} = AMAE_{L1} \times AMAT_{L1} \quad (6)$$

In our exploration algorithm, we first find the L1 cache size that gives us the lowest energy delay product or $min(EDP_{L1})$ in the set of L1 cache sizes that we explore. Once we know the optimal (lowest EDP) L1 cache size, we know the values for $AMAE_{L1}$ and $AMAT_{L1}$. Then, we repeat the same process to find the optimal filter cache size.

4 Analysis of program execution in multitasking applications

In this section, we analyze the nature of control flow in multitasking systems. We start off by examining the execution footprint of a typical multitasking application. Then, we make deductions on the level of determinism in control flow of multitasking applications.

4.1 Execution footprint of a multitasking application

There are inherent differences in the way a task is executed in a single-task based system and a multi-task based system. In single task systems, the system has to do perform only one task and therefore, the lone task runs on the processor all the time. There is no other task to pre-empt the execution of the single-task based application. On the other hand, in multitasking systems, there are multiple tasks being serviced by the processor. They compete with each other for resources like the processor, memory, etc. This creates a complicated situation where there is a need for task arbitration to decides which task gets to execute and a task is often required to be pre-empted before it can finish execution. Such arbitrations to allocate system resources are usually peformed by an operating system.

A typical execution footprint of a multitasking system is shown in figure 4. As can be seen, execution of task A is divided into four sections - A1, A2, A3 and A4. There could be many reasons due to which the control flow may shift to the RTOS or other tasks. The suspension of execution of the tasks could be because it may request to perform a priveleged action which is usually achieved through system calls. In this example, task A may request to read from a file through system call SA1 and the request is serviced by the RTOS. The suspension of task A could also be because of the occurence of a trap or interrupt which needs urgent servicing from the RTOS. An example of such an interrupt due to which the execution of task A needed to be suspended is IT3. Another reason for the suspension of the execution of a task could be because it has already run for a while and the RTOS feels that it should yield the processor to other tasks with higher priorities which are waiting to execute. The RTOS ensures that each task gets a fair share of processor time by keep tabs on the amount of time a process executes. A process is allowed to execute

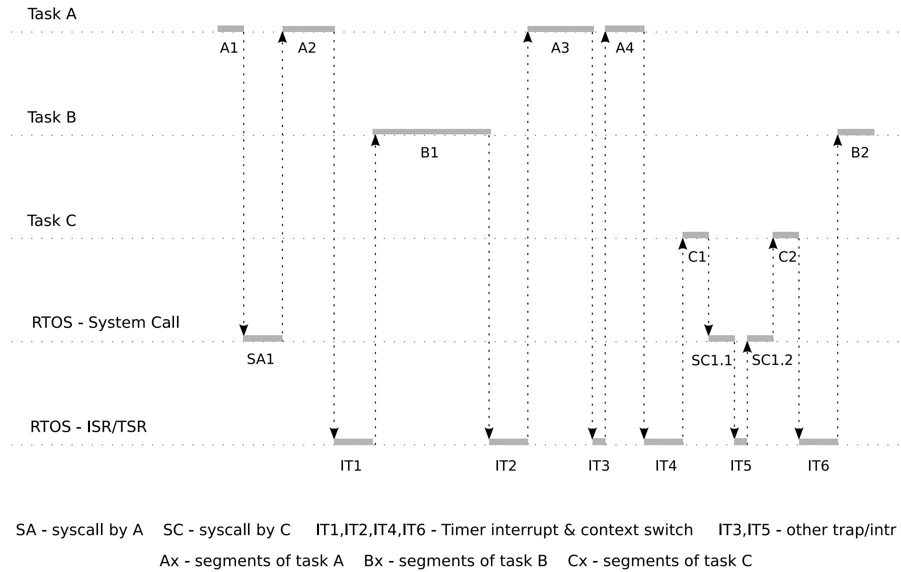


Figure 4: Execution footprint of a multitasking application

one time-quantum at a time called a *timeslice*. After expiration of a timeslice, the scheduler in the RTOS picks the task with the highest priority that is ready to run. The scheduler is also responsible for putting processes to sleep if they request a resource currently held by a higher priority task and also to wake up tasks if a resource that they requested becomes available. In figure 4, the RTOS switches to the higher priority task B after the expiration of the *timeslice* for A at the end of A2.

4.2 Pseudo-random control flow

When an application executes in a single-task based system, the sequence of execution within the application can be analyzed. It is possible to know what are possible sequences of basic blocks that can be executed during the execution of the task based on analysis of the control flow and data flow in the task. For example for task A in 4 , the executions of segments A1, A2, A3, A4 and the system call SA1 are all part of a possible sequence of execution for task A which can be captured using compile time analysis of task A.

Commenting on the sequence of execution in multi-task based systems is however trickier and not always possible. It has already been discussed that in a multitasking system, the execution of individual tasks is interspersed with the execution of other tasks. The order in which the execution is interspersed depends on circumstances which can only be determined at runtime. The pattern of switching amongst tasks is therefore difficult to predict inducing a sense on non-determinism in the execution sequence. A good example of such non-determinism would be the occasional execution of interrupt or trap service routines or switching to other user tasks. The execution of these interrupt or trap service routines is not specified explicitly but they need to be executed every once in a while because they are critical to the execution of the multitasking

system. For example, IT3 and IT5 could very well be critical interrupts that require immediate servicing by the RTOS and therefore require that the current task be abandoned and the interrupt be serviced.

This element of non-determinism conbined with the level of confidence about the expected flow of control in individual tasks makes the execution sequence in multitasking systems pseudo-random. Any technique for analysis of multitasking systems must therefore be sensitive to the pseudo-random nature of control flow in multitasking systems.

5 Impact of pseudo-random control flow on cache accesses

Execution of any application as a single-task based system is very different from its execution in a multitasking system. While a task is executed without interruptions in a single-task based system, its deterministic control flow is sometimes suspended in a multitasking system. This could be because of a plethora of reasons over which the task may or may not have any control. These reasons could be the unavailability of a system resource, competition for processor time with a task having a higher priority, processing of an asynchronous event like an interrupt, trap, etc by the operating system. In the case the task execution is suspended, the multitasking application takes a non-deterministic control path and switches to another task or interrupt service routine. Both deterministic and random control paths in multi-tasking applications have implications on the cache behaviour of the application. The nature and impact of this implication is discussed in this section.

5.1 Intrinsic and extrinisic code interference

Corresponding to the two different types of control flow in multitasking systems, there are two ways in which a cache line could be evicted out of the cache resulting in a cache miss for future references to the same location. We discuss these two kinds misses named as intrinsic misses and extrinsic misses below. Intrinsic interference is the competition amongst the instructions of the same task to occupy the cache. For example, in figure 5, it is possible that BB1 and BB5 occupy the same cache line. In such a case, the contents of BB1 will be evicted from the cache by BB5. This will result in a cache miss the next time BB1 is referenced. Such misses are categorized as missed due to intrinsic interference.

Extrinsic interference on the other hand is the competition amongst different tasks and interrupt service routines to fill the cache with their own instructions. Extrinsic interference is a direct result of the interspersed execution of the tasks in multitasking systems. To explain the cause for extrinsic interference, take the case of figure 5 as an example. In this case, it is likely that some instruction of BB1 that are already present in the cache may be evicted out of the cache if task 'X' is executed. This may lead to cache misses when those instructions are executed again. Such misses are categorized as misses due to extrinsic interference and are in addition to the misses due to intrinsic interference that the application may suffer.

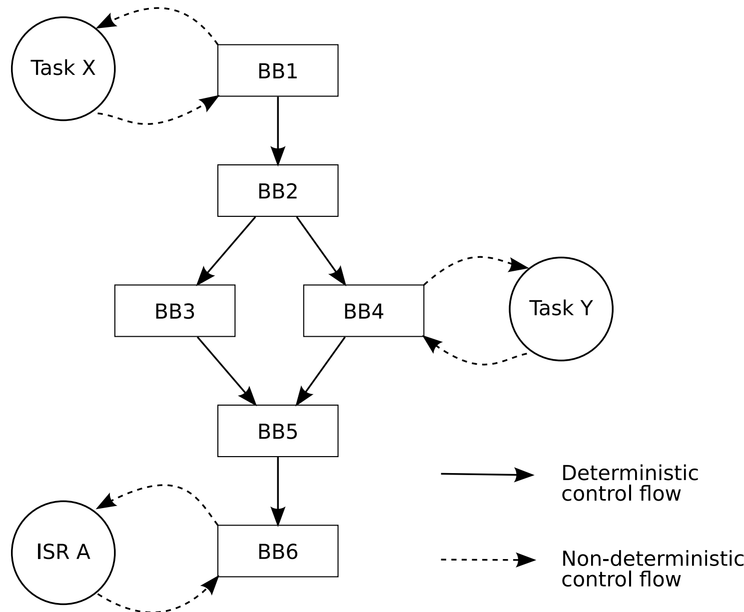


Figure 5: Flow of control in multitasking systems

5.2 Empirical evaluation of cache misses due to code interference

To evaluate the relative frequency of intrinsic and extrinsic misses, we simulated a few multitasking applications and measured the miss rates. In our experimental setup, shown in 6, we connected dedicated cache for each task and housekeeping functions that the RTOS may perform. The number of misses incurred in each of these cache memories is the number of intrinsic misses incurred by the corresponding task or ISR/TSR. If these misses are added up, we can obtain the total number of intrinsic misses incurred if these elements are run together in a multitasking system. We connected the same simulation of the multitasking system to a global cache as well as shown in 6. The number of misses resulting from this cache will include both the intrinsic and extrinsic misses. The extrinsic misses in this case will be the result of cache line evictions due to interference amongst tasks and other RTOS housekeeping functions. As expected, there was a decrease in the number of misses as compared to the first experiment since the second experiment ignored the case of extrinsic interference. Due to the lowered number of misses, the energy delay product for the second experiment was also found to be lower as compared to the first experiment.

Figure 7 shows the miss rate curve and the energy delay for two of eight multitasking programs tested. The difference in the miss rates and hence the energy delay product because of extrinsic interference, however, is found to be insignificantly low. This conclusion is further validated in table 1. The average difference in instruction cache miss rate is only 0.02% to 0.08% of all instructions executed. Consequently, as shown in table 1, the difference in energy delay product is also found to be negligible.

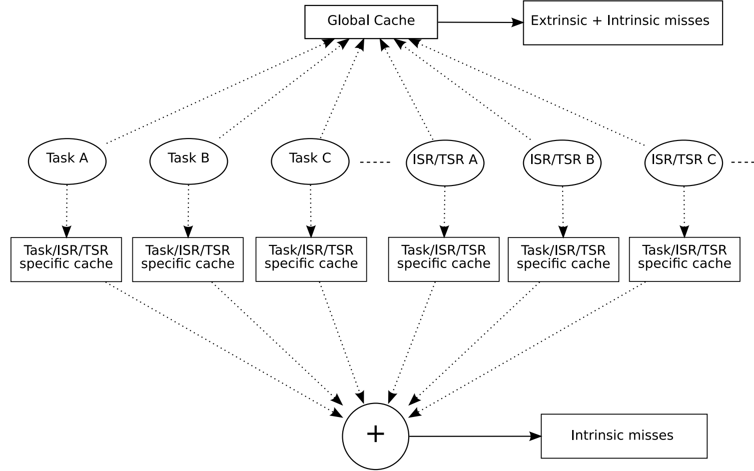
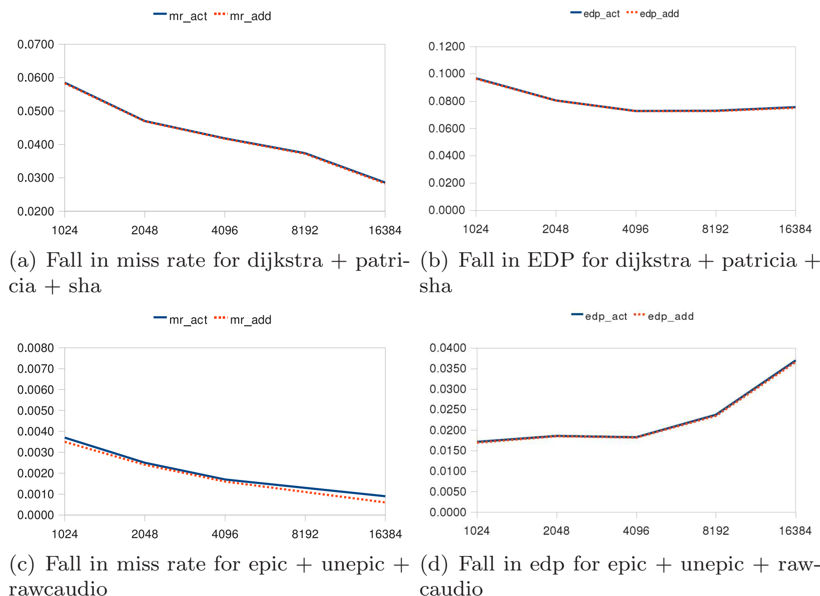


Figure 6: Evaluation of intrinsic and extrinsic interference

Multitasking benchmark	MR increase %	EDP increase %
dijk + patr + sha	0.08	0.37
rawd + cjpeg + sha	0.02	1.01
epic + unepic + rawc	0.02	0.90
djpeg + rawc + sha	0.02	0.96
sha + cjpeg + rawc	0.02	1.03
rawc + rawd + cjpeg + djpeg	0.02	0.86
patr + dijk + rawc + sha	0.02	0.34
dijk + sha + rawc + rawd	0.02	0.70

Table 1: Average increase in miss rate and energy delay product



(a) Fall in miss rate for dijkstra + patricia + sha

(b) Fall in EDP for dijkstra + patricia + sha

(c) Fall in miss rate for epic + unepic + rawcaudio

(d) Fall in edp for epic + unepic + rawcaudio

Figure 7: Difference between effects of standalone task/interrupt simulation added together and the actual multitasking system 7(a), 7(b), 7(c) and 7(d)

5.3 Theoretical explanation of empirical observations

The results obtained can be explained based on the observation that misses due to extrinsic interference happen once in a while as compared to missed due to intrinsic interference. They happen only when the thread of execution switches from one task to another or the other hosekeeping functions that the RTOS may perform like servicing interrupts, traps, etc. As the execution proceeds after the switch, these transient misses incurred become negligible as compared to the rest of the misses incurred due to intrinsic interference or cache line conflicts.

To gauge the relative impact of extrinsic interference as compared to intrinsic interference, consider the example shown in figure 8. Here, the cache contents are shown to contain a loop consisting of four basic blocks - BB1, BB2, BB3 and BB4. However, during the execution of the loop, it is pre-empted by 'X' which could be a another task or an interrupt service routine. When the loop resumes execution after 'X' has finished executing, some of the instructions belonging to the loop would have been evicted. But, after just one iteration, the cache is filled up again with insructions of the loop and subsequent accessess to the instructions of the loops are hits.

Intrinsic interference in the example shown in figure 8 depends on the frequency of the loop shown in figure 8 which is significantly higher than the number of pre-emptions that might happen during the lifetime of the loop. It is therefore expected that the number of misses due to pre-emption or extrinsic interference will be much lesser as compared to the number of misses due to intrinsic interference. Considering that embedded applications spend a significant amount of time in such loops, it can be safely assumed that the frequency of these loops will be very high as compared to the misses due to the occasional pre-emptions of those loops as shown in this subsection. Consequently,

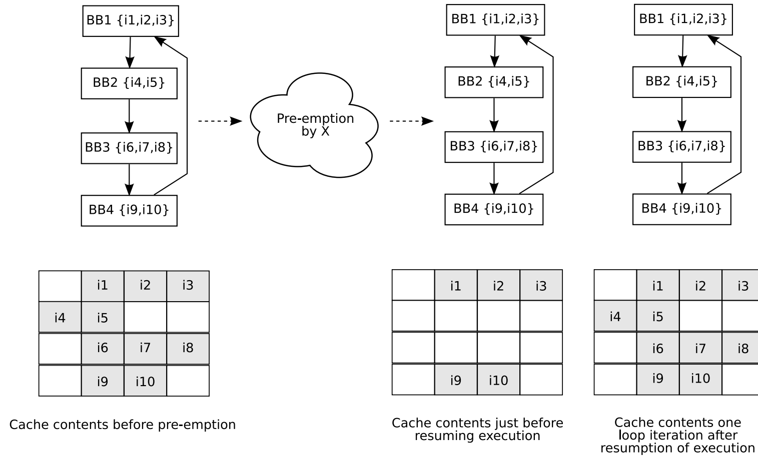


Figure 8: Effect of extrinsic interference

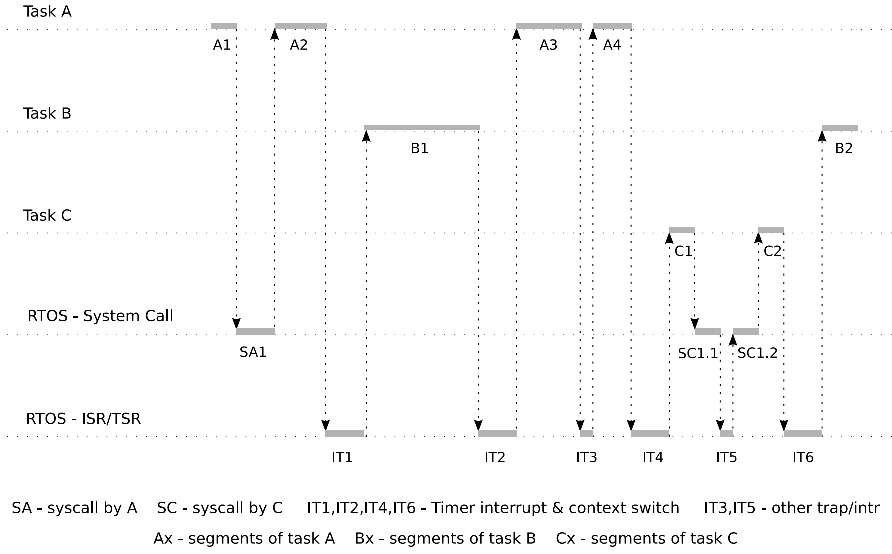
the damage done by intrinsic interference is much higher as compared to extrinsic interference and therefore, intrinsic interferenceforms can be treated as the main driving force behind choosing an appropriate cache for the application using the cache tuning heuristics.

6 Proposed technique for cache tuning for multitasking systems

In the previous section, we discussed the relative impact of extrinsic interference and intrinsic interference on the total cache hit/miss rate. We showed that misses due to extrinsic interference to be significantly lesser than those due to intrinsic interference. In this section, we develop ideas for a cache tuning framework for multitasking systems based on these observations.

As a task executes in a multitasking environment, the instructions of the loops would be flushed once in a while by intervening tasks or interrupts. However, the loops themselves will execute to completion regardless of the number of times they are pre-empted. The pre-emption might result in a few additional misses as shown in the preceeding section but it should not affect the optimal cache size for the application provided all the loops and their relative frequencies are taken into account in choosing the cache size.

To make this point clear, consider a two task system as an example. Task 1 (T1) has a loop of size 1kB which executes an aggregate of 1000 times. Lets assume that its execution is interspersed with another task 2 (T2) which does not have any loops in it. In this case, an ideal cache size for the application would be 1kB because it can hold the loop in T1 and mitigate any unnecessary cache misses. However, if there is another loop in T2 which has a size of 4kB and a significant frequency, then the optimal cache size for the system would be between 1kB and 4kB. The exact size of the optimal cache would depend on



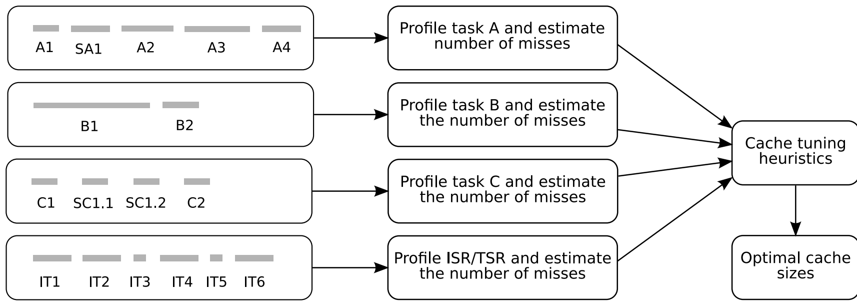


Figure 9: Cache tuning methodology for multitasking systems

the frequency of the loop in T2. If the frequency is too low, then a cache size of 1kB mght be just right for the application. On the other hand, if it is too high, it would be wise to choode a cache size of 4kB. The choice of the optimal cache would be independent of how the execution of T1 and T2 is interspersed with each other because regardless of how many times T1 and T2 pre-empt each other, the loops in them would execute to completion.

Regardless of the order in which the tasks execute in a multitasking environment, the sole factor that determines the optimal cache size is the size and frequency of all the loops inherent in the tasks. In the following subsections, we present the broad methodology of our approach to profiling multitasking systems for cache tuning and the cache tuning framework itself. Our techniques rely on simulating the multitasking application to obtain information about the loops using loop profiling techniques discussed earlier. For all purposes, we ignore the order in which tasks execute and just focus on obtaining relative frequencies of the loops in the multitasking application.

6.1 Loop profiling of user tasks

In a multitasking environment, the operating system is responsible for process, memory and I/O management. This is different from the execution of a standalone task wherein the task has absolute control over memory and I/O and can read and write to and from these entities as and when it wishes. Take as an example, the case of writing to memory. In the case of a standalone task, this operation proceeds without any checks on whether the task is permitted to write to the memory location or whether the memory resource is free in the first place. However, in a multitasking environment running under the control of an operating system, the tasks can write to a memory only through a system call by invoking the operating system. In a system call, the control is handed over to the operating system which follows a pre-defined set of checks and protocols before it allows the write to proceed.

Such system calls are often invoked from within loops in user tasks. Any technique for loop profiling of user level tasks must take into account any system call that the loop may invoke. The size of the system call(s) added to the size of the loop gives an accurate indication of the size of the cache required to hold the loop. For this reason, we profile user level tasks by taking into account any system calls that they might invoke. Take the example of figure 9, as task A executes, it makes system call SA1. This system call could very well be invoked from within a loop. Therefore, it is imperative to consider system calls as a part of loops in user tasks.

To identify the nature of loops user task, we simulate the multitasking application but profile individual tasks as if they were the only task running on top of a RTOS as shown in 9. Thereby, we ignore any pre-emptions that it might suffer from other tasks or other RTOS house-keeping functions like task management and interrupt/trap handling but ensure that the system calls are taken into account while calculating loop sizes for the task. This technique will gives us an indication of the intrinsic misses that the multitasking application is likely to incur because of this task. If there are multiple tasks running on the system, this exercise can be repeated for every task executing on the system.

6.2 Loop profiling of the RTOS

Taking the tasks into consideration alone is not enough. Locality analysis for cache tuning must also take into account any loops that are executed by the operating system for house-keeping purposes. In a RTOS based multitasking system, the RTOS is the manager of processes, memory, I/O, etc. When executing multiple tasks, there are bound to be overheads in executing the RTOS code that is responsible arbitrating multiple tasks competing for resources. For example, one of these tasks is to maintain timers to monitor how long a task has been running. For this, it has to respond to interrupts from the hardware timer periodically to update process times. Once a task exceeds the timeslice allocated to it, the RTOS is reponsible for putting it to sleep and selecting a new process to run based on a scheduling algorithm. Task switching in this case would involve saving the context of the current task and loading the context of a new task.

Another one of such tasks performed by the RTOS would be to respond to page faults. In multiasking systems, because the memory is shared by several

tasks, every task is given the impression that it is working with large, contiguous sections of memory. In reality, each process' memory may be dispersed across different areas of main memory, or to secondary storage. When a process requests access to its memory, virtual addresses are mapped to physical addresses through a page table. Sometimes it so happens that a physical memory frame may not have been allocated for the virtual address. In such a case, there will be a page fault and the operating system is responsible for dealing with it. The handler for the page fault allocates a physical memory frame to the virtual page and create a page table entery and the memory access will continue.

It is evident from the above that the RTOS needs to do a lot of house-keeping functions when multitasking applications run on it because all system resources are shared by multiple tasks. Therefore, for tuning the cache for multitasking systems, it is imperative to consider the loops in the house-keeping functions that the RTOS performs. These loops will be in addition to any user task level loops or loops in system calls. To identify these loops, we simulate the multitasking application and profile the RTOS components responsible for performing these functions independent to profiling the tasks. This approach is shown in figure 9 wherein IT1 to IT6 are profiled as being part of the RTOS housekeeping functions.

6.3 Overview of the tuning framework

The overall framework for tuning the cache for multitasking system is shown in figure 10. The framework is divided into three major components. In the first stage, we simulate the multitasking system to obtain the call graph and control flow graphs of the individual tasks, the associated system calls and other house-keeping functions that the RTOS may perform for the execution of the multitasking application. After obtaining the call graph and the control flow graphs, we profile them on a task by task basis to obtain the nature of the loops inherent in them. We also profile the RTOS for obtaining the loops in all the house-keeping functions that it may perform. After obtaining information about all loops, we use this information to estimate the hit rates for multiple cache sizes in one go. In the final stage, we feed the estimated hit rates to a tuning engine which performs cache tuning based upon a predefined set of heuristics. Because we perform energy-centric cache tuning of the predictor based filter cache hierarchy in this work, this module also uses the cache energy per access obtained from the CACTI models. The final output of the framework is the optimal filter cache and optimal L1 cache size i.e. the cache sizes which results in the lowest energy delay product for the application.

7 Experimental results

In this section, we present our results on cache tuning for multitasking applications. We first discuss our experimental setup and then go on to compare design space exploration using our proposed techniques with design space exploration using exhaustive simulation.

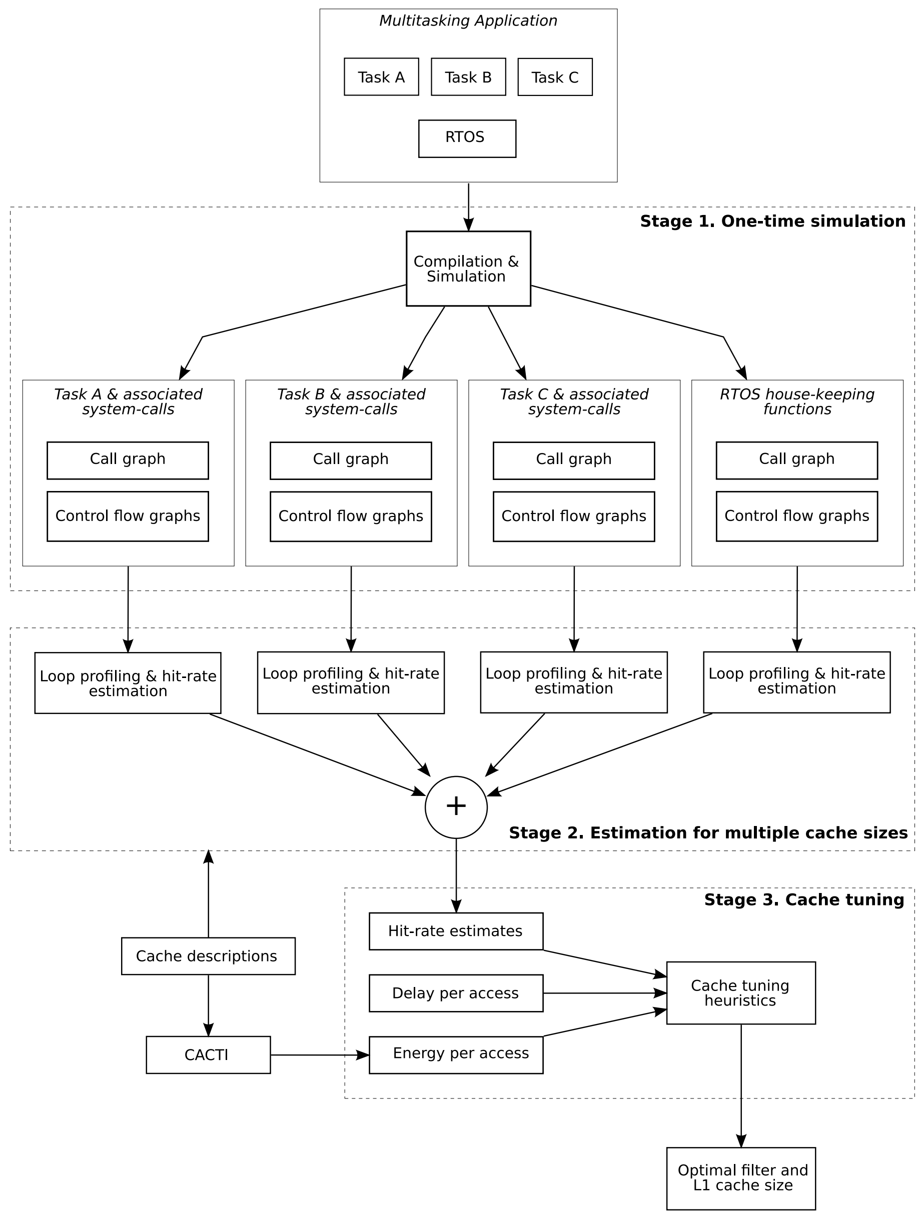


Figure 10: Cache tuning framework for multitasking systems

Cache parameter	Value or range
FC size range	256B to L1 Size
FC block size	16B
FC associativity	1
FC access time	1 cycle
L1 size range	1kB to 32kB
L1 block size	32B
L1 associativity	4-way
L1 access time	1 cycle
L2 size	512kB
L2 block size	32B
L2 associativity	4-way
L2 access time	6 cycle

Table 2: Instruction cache parameters for design space exploration

7.1 Experimental Setup

We ran the benchmark programs from the Mibench [13] and the Mediabench [17] benchmark suite on the M5 simulator [5]. The M5 simulator simulated a single issue processor with the Alpha Dec instruction set. The operating system running on the simulator was Linux. We assumed a unified L2 cache of 512kB with a 32B block size and 4-way set associativity. The selection space for the design space exploration and the instruction cache parameters used are shown in table 2. The measurements of access energy for the different cache configurations were obtained using the CACTI tool [36].

7.2 Comparison of proposed techniques with exhaustive simulation

The proposed EDP estimation techniques closely follow the results obtained through exhaustive simulation of the instruction cache hierarchy. Table 3 shows the filter cache and L1 cache sizes chosen through exhaustive simulation (sim) and through the estimation heuristics (est). The exhaustive simulation approach is always able to identify the optimal configuration - the one with the least EDP - because it resorts to brute force search. However, our estimation heuristics are also able to identify the optimal configuration or a near-optimal one in all cases. The EDP of the configurations chosen by the two techniques is shown (in figure 11) next to the EDP of a base 32kB, 4-way set associative and 32B line size L1 cache which is the norm in many processors. As is evident from the figure, the EDP of the configuration obtained through the estimation techniques is optimal in almost all cases and near-optimal for the rest.

While the estimation technique has been shown to identify optimal or near-optimal cache configurations in almost all cases, the real value of the proposition comes through in the runtime of the algorithm and its scalability.

Table 4 shows the time that was required to run the estimation algorithms as compared to an exhaustive simulation. As can be seen, a significant amount of time is saved if the design space exploration is done using the estimation heuristics. For some of the multitasking programs, the estimation techniques are able to achieve a speedup of upto an order of magnitude.

Multitasking benchmark	Config(est)	Config(sim)
dijk + patr + sha	(1kB,32kB)	(1kB,32KB)
rawd + cjpeg + sha	(512b,4kB)	(512B,4kB)
epic + unepic + rawc	(256B,1kB)	(512B,1kB)
djpeg + rawc + sha	(512B,4kB)	(1kB,4kB)
sha + cjpeg + rawc	(512B,4kB)	(512B,4kB)
rawc + rawd + cjpeg + djpeg	(1kB,4kB)	(1kB,4kB)
patr + dijk + rawc + sha	(1kB,32kB)	(1kB,32kb)
dijk + sha + rawc + rawd	(256B,4kB)	(512B,4kB)

Table 3: Filter cache and L1 cache sizes chosen by the estimation algorithm and the exhaustive simulation

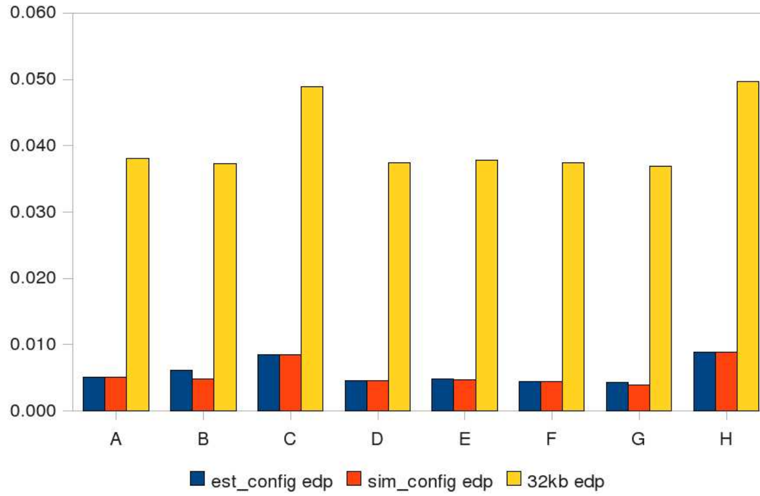


Figure 11: Savings in EDP achieved by estimation algorithm vs. exhaustive simulation

Multitasking benchmark	Time(est)	Time(sim)	Savings
dijk + patr + sha	6m 29s	35m 17s	82%
rawd + cjpeg + sha	4m 15s	11m 18s	62%
epic + unepic + rawc	2m 44s	23m 50s	89%
djpeg + rawc + sha	1m 56s	9m 30s	80%
sha + cjpeg + rawc	1m 56s	13m 13s	85%
rawc + rawd + cjpeg + djpeg	3m 35s	13m 17s	73%
patr + dijk + rawc + sha	11m 56s	39m 39s	70%
dijk + sha + rawc + rawd	5m 59s	18m 1s	67%

Table 4: Comparison of the average runtime for the estimation algorithm and exhaustion simulation techniques

Another huge advantage of such an approach is its scalability. The time complexity of the exhaustive simulation techniques is O(n) where n is the number of configurations tested. On the other hand, the time complexity of the estimation heuristics is constant irrespective of the number of cache sizes that the estimation is required for. This constant time is the time taken to simulate the application, build its call graph and control flow graph and to estimate the EDP for different cache sizes through a one-time loop profiling.

8 Estimation of misses due to extrinsic interference

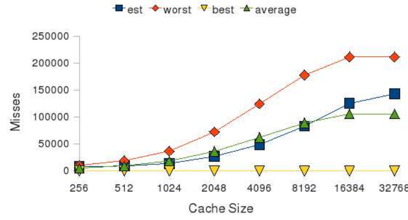
While the number of misses due to extrinsic interference or code mixing have been shown to be nominal in the preceeding section. It is useful to have a way to estimate the number of such misses. In this section, we describe a technique to estimate the number of misses due to code mixing. Our approach is based on estimating the worst-case and best-case misses due to code mixing and then taking the average of the best and the worst case as the estimate. In the subsequent part of this section, we first describe how to obtain the worst-case estimate, then we discuss what is the best-case estimate and finally conclude the section by show that the average-case estimate follows the actual number of misses incurred due to code mixing closely.

We start off the estimation process by estimating a worst-case estimate for the number of misses that could happen due to code mixing. We assume the following for our worst-case estimate. Firstly, we assume that when interrupts or context switches occur, they flush out useful cache lines that could have been potentially been reused at a later time. Secondly, the number of lines that are flushed out is equal to the number of cache lines that the interrupt service routine occupies in the cache if the size of the interrupt service routine is smaller than the cache size. Otherwise, we assume that the number of cache lines flushed out is the total number of cache lines in the cache. This information is shown in equation 7. For the case of context switches, we assume that the cache context is completely flushed out as shown in 8.

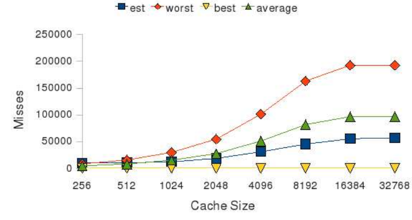
A value for the worst-case estimate of misses due to extrinsic interference can thus be obtained using equation 9.

$$
\begin{aligned}
damage_{intr} &= max(intr_lines, cache_lines) && (7)\\
damage_{ctxsw} &= cache_lines && (8)\\
misses_{ei-wc} &= \sum freq_{intr} \times damage_{intr} \\
&+ \sum freq_{ctxsw} \times damage_{ctxsw} && (9)
\end{aligned}
$$

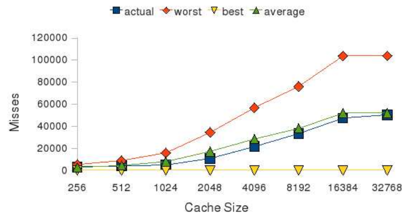
This is a worst-case estimate because the actual number of misses for will be lesser than this. This could be because of two factors - the size of the cache and the timing of the interrupts. If the cache is large enough, the amount of flushing would be lesser than what has been assumed because a large cache would be able to accommodate the instructions of the interrupt service routine without flushing out other cache lines. Also, if interrupts happen in close succession



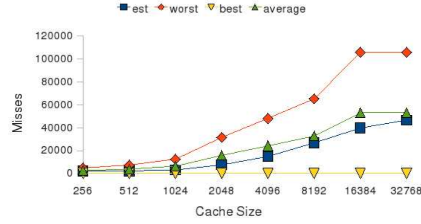
(a) Code mixing misses for dijkstra + patricia + sha



(b) Code mixing misses for epic + unepic + rawcaudio



(c) Code mixing misses for rawcaudio + cjpeg + sha



(d) Code mixing misses for rawcaudio + djpeg + sha

Figure 12: Estimated and actual misses due to extrinsic interference shown in sub-figures 12(a), 12(b), 12(c) and 12(d)

then there is also a chance that the cache lines will be reused instead of being flushed out.

In the best case, it is assumed that every cache line will be fully reused before being flushed out due to any intervening interrupt or context switch. Therefore, there would be no misses due to code mixing. The only misses that are incurred would be a one-time penalty of bringing in the cache lines into the cache and misses due to intrinsic interference. These misses are already captured in the estimated number of misses obtained through standalone task profiling.

As can be seen from figure 12, our average case estimate for the number of misses due to code mixing follow the measured number of misses due to code mixing quite closely. Therefore, the proposed techniques can be relied upon to estimate the number of extrinsic misses in a multitasking application.

9 Conclusion

In this paper, we analyzed the control flow in embedded multitasking systems for the purpose of loop profiling and subsequent cache tuning. We showed that although the nature of control flow in multitasking system is pseudo-random, for cache tuning purposes the interspersed execution of user tasks can be ignored and they can be profiled in isolation to obtain information about the nature of the loops in them. This information about the loops could then be used to identify the optimal cache size for an application. We validated the proposed technique by applying it to tune the cache hierarchy for embedded multitasking applications composed of commonly used embedded benchmarks programs. We divided the system execution into the execution of user-tasks/system-calls and

other housekeeping function that the RTOS may perform. We profiled these components in isolation, obtained information about all the loops that were executed and were able to identify optimal or near-optimal L1 and filter cache size for all multitasking applications considered. Our proposed framework was shown to achieve significant speed-up as compared to exhaustive cache simulations was well. The proposed techniques was shown to be highly scalable as well because it was be relied upon to estimate the energy delay product estimates for any range of cache size after a one time profiling.

References

[1] A. Agarwal, J. Hennessy, and Horowitz M. An analytical cache model. *ACM Transactions on Computer Systems*, 7(2), 1989.

[2] D.H. Albonesi. Selective cache ways: on-demand cache resource allocation. In *32nd Annual International Symposium on Microarchitecture*, pages 248–259, 1999.

[3] Altera. Nios embedded processor 32-bit programmer's reference manual, 2003.

[4] B. T. Bennett and V. J. Kruskal. Lru stack processing. *IBM Journal of Research and Development*, 19:353–357, 1975.

[5] Nathan L. Binkert, Ronald G. Dreslinski, Lisa R. Hsu, Kevin T. Lim, Ali G. Saidi, and Steven K. Reinhardt. The m5 simulator: Modeling networked systems. *IEEE Micro*, 26(4):52–60, 2006.

[6] Steve Carr, Steve Carr, Ken Kennedy, and Ken Kennedy. Compiler blockability of numerical algorithms. In *In Proceedings of Supercomputing '92*, pages 114–124, 1992.

[7] C. K. Chow. Determination of cache's capacity and its matching storage hierarchy. *IEEE Transactions on Computers*, 25(2):677–688, 1976.

[8] Stephanie Coleman and Kathryn S. McKinley. Tile size selection using cache organization and data layout. In *PLDI*, pages 279–290, 1995.

[9] Dennis Gannon, William Jalby, and Kyle Gallivan. Strategies for cache and local memory management by global program transformation. *J. Parallel Distrib. Comput.*, 5(5):587–616, 1988.

[10] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Cache miss equations: An analytical representation of cache misses. In *In Proceedings of the 1997 ACM International Conference on Supercomputing*, pages 317–324. ACM Press, 1997.

[11] Tony Givargis and Frank Vahid. Platune: a tuning framework for system-on-a-chip platforms. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 21(11):1317–1327, 2002.

[12] Ann Gordon-Ross and Frank Vahid. A self-tuning configurable cache. In *DAC '07: Proceedings of the 44th annual conference on Design automation*, pages 234–237, New York, NY, USA, 2007. ACM.

[13] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and T. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *4th IEEE International Workshop on Workload Characteristics*, pages 3–14, December 2001.

[14] M. D. Hill and A. J. Smith. Evaluating associativity in cpu caches. *IEEE Trans. Comput.*, 38(12):1612–1630, 1989.

[15] S. Laha, J. H. Patel, and R. K. Iyer. Accurate low-cost methods for performance evaluation of cache memory systems. *IEEE Trans. Comput.*, 37(11):1325–1336, 1988.

[16] Monica S. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. In *In Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, 1991.

[17] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *International Symposium on Microarchitecture*, pages 330–335, 1997.

[18] Afzal Malik, Bill Moyer, and Dan Cermak. A low power unified cache architecture providing power and performance flexibility (poster session). In *ISLPED '00: Proceedings of the 2000 international symposium on Low power electronics and design*, pages 241–243, New York, NY, USA, 2000. ACM.

[19] R. L. Mattson, J. Gecsei, D. Slutz, and I.L. Traiger. Evaluation techniques for storage hierarchies. *IBM System Journal*, 9(2):78–117, 1970.

[20] Kathryn McKinley and Steve Carr. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18:424–453, 1996.

[21] F. Olken. Efficient methods for calculating the success function of fixed space replacement policies. Technical Report LBL-12370, Lawrence Berkeley Laboratory, 1981.

[22] Maurizio Palesi and Tony Givargis. Multi-objective design space exploration using genetic algorithms. *Hardware/Software Co-Design, International Workshop on*, 0:67, 2002.

[23] Thomas Roberts Puzak. *Analysis of cache replacement-algorithms*. PhD thesis, 1985.

[24] Gabriel Rivera and Chau wen Tseng. Data transformations for eliminating conflict misses. In *In Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 38–49, 1998.

[25] R. Sugumar and S. Abraham. Efficient simulation of multiple cache configurations using binomial trees. Technical report, 1991.

[26] Rabin Andrew Sugumar. *Multi-configuration simulation algorithms for the evaluation of computer architecture designs*. PhD thesis, Ann Arbor, MI, USA, 1993.

[27] Tensilica. Xtensa processor generator, 2005.

[28] D. Thiebaut. On the fractal dimension of computer programs and its application to the prediction of the cache miss ratio. *IEEE Transactions on Computers*, 38(7), 1989.

[29] Er V. Veidenbaum, Weiyu Tang, Rajesh Gupta, Ru Nicolau, and Xiaomei Ji. Adapting cache line size to application behavior. In *In ICS*, pages 145–154, 1999.

[30] Kugan Vivekanandarajah. *Customizable instruction cache hierarchy for embedded systems*. PhD dissertation, Nanyang Technological University, 2005.

[31] Kugan Vivekanandarajah and Thambipillai Srikanthan. Custom instruction filter cache synthesis for low-power embedded systems. In *RSP '05: Proceedings of the 16th IEEE International Workshop on Rapid System Prototyping (RSP'05)*, pages 151–157, Washington, DC, USA, 2005. IEEE Computer Society.

[32] Kugan Vivekanandarajah, Thambipillai Srikanthan, Saurav Bhattacharyya, and Prasanna Venkatesh Kannan. Incorporating pattern prediction technique for energy efficient filter cache design. In *IWSOC*, pages 44–47, 2003.

[33] Kugan Vivekanandarajah, Thambipillai Srikanthan, and Christopher T. Clarke. Profile directed instruction cache tuning for embedded systems. In *ISVLSI '06: Proceedings of the IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures*, page 277, Washington, DC, USA, 2006. IEEE Computer Society.

[34] Kugan Vivekanandarajah, Thambipillai Srikanthan, Christopher T. Clarke, and Saurav Bhattacharyya. Static pattern predictor (spp) based low power instruction cache design. In *Embedded Systems and Applications*, pages 210–215, 2003.

[35] Wen-Hann Wang and Jean-Loup Baer. Efficient trace-driven simulation method for cache performance analysis. In *SIGMETRICS '90: Proceedings of the 1990 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 27–36, New York, NY, USA, 1990. ACM Press.

[36] S. Wilton and N. Jouppi. Cacti: An enhanced cache access and cycle time model, 1996.

[37] Zhao Wu and Wayne Wolf. Iterative cache simulation of embedded cpus with trace stripping. In *CODES '99: Proceedings of the seventh international workshop on Hardware/software codesign*, pages 95–99, New York, NY, USA, 1999. ACM Press.

[38] Xilinx. Microblaze hardware reference guide.

[39] C. Zhang and F. Vahid. Cache configuration exploration on prototyping platforms. In *Proceedings of International Workshop on Rapid Systems Prototyping*, 2003.

[40] Chuanjun Zhang, Frank Vahid, and Roman Lysecky. A self-tuning cache architecture for embedded systems. In *Design, Automation and Test Conference in Europe (DATE), 2004*, pages 142–147, 2004.

[41] Chuanjun Zhang, Frank Vahid, and Walid A. Najjar. A highly configurable cache for low energy embedded systems. *ACM Transactions in Embedded Computing Systems*, 4(2):363–387, 2005.