ABSTRACT
Modern embedded systems execute a single application or a class of applications repeatedly. A new emerging methodology of designing embedded systems utilizes configurable processors where the cache size, associativity, and line size can be chosen by the designer. In this paper, a method is given to rapidly find the L1 cache miss rate of an application. An energy model and an execution time model are developed to find the best cache configuration for the given embedded application. Using benchmarks from Mediabench, we find that our method is on average 45 times faster to explore the design space, compared to Dinero IV while still having 100% accuracy.

1. Introduction
Today, cache memory is an integral component of mid to high end processor based embedded systems. The inclusion of cache significantly improves system performance and reduces energy consumption. Current processor design methodologies rely on reserving large enough chip area for caches while conforming with area, performance, and energy cost constraints. Recent application specific processor design platforms (such as the Tensilica’s Xtensa platform [1]) allows a cache to be customized for the processor. This allows a designer which can meet tighter energy consumption, performance, and cost constraints.

In existing low power processors, cache memory is known to consume a large portion of the on-chip energy. For example, in [2] Montanaro et al. report that cache consumes up to 43% of the total energy of a processor. In embedded systems where a single application or a class of applications are repeatedly executed on a processor, the memory hierarchy could be customized such that an optimal configuration is achieved. The right choice of cache configuration for a given application could have a significant impact on overall performance and energy consumption.

Choosing the correct cache configuration for an embedded system is crucial in reducing energy consumption and improving performance. To find the correct configuration the hit and miss rates must be evaluated, and the resulting energy consumption and execution times must be accurately estimated. Estimating the hit and miss rates (for a particular application with sample input data) is fairly easy using tools such as Dinero IV [7], but enormously time consuming to do so for various cache sizes, associativities and line sizes. The resulting energy consumption and execution times are difficult to examine due to the non uniform nature of memory, where the first memory access takes far greater time compared to subsequent accesses (which are sequential to the first access). Energy and access times are further complicated by differing cache configurations consuming energy at different rates and taking differing amounts of time to access.

Our research results demonstrate that low miss rates do not necessarily mean a faster execution time. Figure 1 shows the effect of different cache configurations have on the number of total cache misses and the total execution time for the G721 encode application. The graph in Figure 1 shows that higher total cache miss rates can possibly provide the fastest execution time. This is due to large or more complex (higher associativity) caches having significantly longer access times.

ABSTRACT
Finding Optimal L1 Cache Configuration for Embedded Systems
Andhi Janapsatya†, Aleksandar Ignjatović‡, Sri Parameswaran††
†School of Computer Science and Engineering, The University of New South Wales
Sydney, NSW 2052, Australia
‡NICTA, The University of New South Wales
Sydney, NSW 2052, Australia
{andhij,sridevan,ignjat}@cse.unsw.edu.au

Figure 1: Total Cache miss vs. Total Execution Time

0-7803-9451-8/06/$20.00 ©2006 IEEE.
796
2. Related Work

Cache simulation is required for tuning cache parameters to ensure maximal performance and minimal energy consumption. In the past, various methodologies have been researched for cache simulation. These cache simulation methodologies can be divided into two classes: one, estimation techniques, and the other, exact simulation.

Cache estimation techniques use heuristics to predict the cache misses for multiple cache configurations. Pieper et al. in [3] developed a metric to represent cache behavior independently of the cache structure. Their metric based result is within 20% accuracy of a uniprocessor trace-based simulation and can be applied for estimating multiprocessor architectures. In [4], Fornaciari presented a heuristic method for configuration of cache architecture without exhaustive analysis of the space of parameters. Their analysis looked at the sensitivity of individual cache parameters on the energy delay product. Maximum error is less than 10%.

Ghosh et al. described a method to generate and solve a cache miss equations (CME) to represent the cache behavior [5]. In [6], Vera et al. proposed a fast and accurate method to solve the cache miss equation (CME).

For exact cache simulation techniques, there exists a tool called Dino IV [7]. Dino IV is a single processor cache simulation tool developed by Jan Edler and Mark Hill. Its purpose is to estimate the number of cache misses given a cache configuration; its features include simulating separate or combined instruction and data caches, and simulating multiple levels of cache.

For simulating multiple cache configurations, exact cache simulation techniques rely on exploiting the inclusion property of caches. Inclusion means that given two cache configurations, Cache \( C_2 \subseteq Cache \ C_1 \) if all the content of Cache \( C_2 \) is a subset of the content of Cache \( C_1 \).

In 1970, Gecsei et al. [8] introduced the ‘Stack’ algorithm for performing simulation of multiple levels of storage systems. In 1989, Hill in [9] investigated the effects of associativity of caches. They briefly described the methodology of forest simulation for quick simulation of alternate direct-mapped caches. They also introduced the all-associative methodology for simulating alternate direct-mapped, set-associative, and fully-associative caches based on the ‘Stack’ algorithm. The space complexity of the all-associativity simulation is \( O(N_{unique}) \), where \( N_{unique} \) is the number of unique blocks referenced in an address trace. In their experiment, they showed that to simulate alternate direct-mapped caches, the forest simulation method is faster than the all-associative simulation.

Sugumar et al., introduced a cache simulation methodology by using a binomial trees [10] to improve the method described in [9]. The time complexity of their algorithm for searching procedure is \( O((\log_2(N) + 1) \times A) \) and the time complexity for maintaining the binomial tree is \( O((\log_2(N) + 1) \times A) \), where \( N \) is the size of the cache and \( A \) is the associativity of the cache. Li in [11] then extends the work in [10] by introducing a method to compress the program trace for reducing the cache simulation time.

Other existing exact cache simulation methodologies uses parallel processing units and/or multiprocessor systems. Nicol in [12] presented a parallel methodology to simulate cache using SIMD and MIMD hardware units.

Heidelberger in [13] presented a method of to analyze a cache trace using a parallel processor system. They split long traces into several shorter traces, and the shorter traces are then executed on parallel independent processors. The sum of the individual results are not accurate, but by executing a re-simulation phase, it is possible to accurately count the exact number of cache misses.

Our simulation methodology was created as a forest simulation data structure. Our methodology extends the idea of forest simulation described in [9]. The space complexity of our methodology is fixed depending on the number of cache configurations to be evaluated. The required space of our cache simulation method is larger than the space needed for the all-associativity simulation described in [9] and the binomial tree simulation described in [10]. The time complexity for searching our data structure is \( O((\log_2(N) + 1) \times A) \) and the time complexity for updating the data structure is \( O((\log_2(N) + 1) \times A) \). This is faster compared to the method described in [10].

Other research looked at the use of heuristics for predicting cache behavior. Ghosh et al. in [14] presented an algorithm for simulating cache parameters and finding cache configurations that guaranteed cache miss rates lower than a desired cache miss rate. Their space complexity is in the order of the size of the trace file.

2.1 Our Contribution

- For the first time a methodology is proposed which allows an L1 cache to be chosen for an Application Specific Instruction Set Processor (ASIP) based upon the energy consumption and execution time.
- We also propose a modified forest algorithm based on a simplified data structure, for fast and accurate simulation of the cache. The time taken by the algorithm for both simulation and updating of the data structure is considerably quicker than previous methods.

In addition, this method allows parallelization.

3. Cache Parameters Exploration Methodology

A cache configuration is dependent on the cache parameters: cache size \( N \), cache associativity \( A \), and cache line size \( L \). The cache size refers to the total number of bits that can be stored in the cache. The cache associativity refers to the number of ways a data can be stored within the same address of the cache. For a direct-mapped cache \( (A = 1) \), each datum has a single location where it can be stored within the cache. The total cache size divided by associativity of the cache is called the cache set size, \( M = N/A \). In our simulations we will consider cache configurations with the cache set in the range from \( 2^{m_{min}} \) to \( 2^{m_{max}} \), where \( m_{min} \) and \( m_{max} \) refer to the number of address bits needed to address \( 2^{m_{min}} \) and \( 2^{m_{max}} \) locations.

We perform design space exploration on the cache parameters by accurately and efficiently simulating the number of cache misses that would occur for a given collection of cache configurations. We optimize the run time of our cache simulation by replacing multiple readings of large program traces with a single reading and simulating multiple cache configurations simultaneously. This is possible due to the following observations.

First, assume that two cache configurations have the same associativity and the same cache line size but that one cache is twice the size of the other cache. In this case if a cache hit occurs on the smaller size cache, a cache hit will also occur on the larger cache size.

This is illustrated in Figure 2. For a memory address request of ‘1010’, cache location pointed to by the address ‘1010’ can be found on the cache locations shown with the dotted line branches in Figure 2. The numbers inside the parentheses shown in Figure 2 indicate the cache address for that location. From the figure, it can be seen that the entries within cache size = 2 are a subset of the entries within cache size = 4, and the entries within cache size = 4 are a subset of the entries within cache size = 8.

The second observation is that if a cache miss occurs for a cache of size \( N \), associativity \( A \), and of cache set size \( M \), then for all other cache configurations with the same cache set size \( M \) and associativity larger than \( A \), a cache hit will also occur. Hence, evaluation for all values larger than \( A \) is not required to determine the number of cache
For each address $x$ from the trace
Use the least significant $m_{\text{min}}$ bits of the address $x$ to locate in the array, the pointer to the corresponding tree, and go to the root of the tree.
For $k = 0$ to $(m_{\text{max}} - m_{\text{min}})$ \{ if $k$ corresponds to the level of the tree; \}
Go to the head of the linked list pointed by this node of the tree
Search linked list to find the tag entry that is equal to the tag of the address $x$.
If a cache hit occurs in an element $s$ of the linked list{ Increment cache miss counter for all cache configuration with cache set size equal to $2^k$ and lower associativity than $s$. Move element $s$ to be the head element of the linked list.} else \{ Increment cache miss counter for all cache configuration with cache set equal to $2^k$. Replace entry in the tail element of the linked list with the current tag address. Move this tail element to become the head element of the linked list. } Increment the value of $k$. Step down the tree according to the value of the bit $k + m_{\text{min}}$, taking the left child if this bit is 0 else take the right child.
} Scan the next address $x + 1$ from the trace.

tree depending on the $m_{\text{min}}$ to $m_{\text{max}} + k$ bits of the address to choose whether to traverse the left branch or the right branch. The node has a pointer to the head element in the linked list that corresponds to the multiple way associativity of this cache set location. The remaining bits of the address $x$ form the tag address that is to be searched for in the linked list. If the tag is found at position $x$, this indicates that a cache hit would occur in all $s$-way or higher-way associative caches. If this happens, the $s^{th}$ element of the list is moved to the head of the list. In this case, all cache miss counters for caches with the same cache set size and associativity less than $s$ are incremented. If the tag is not found, the tag in the tail element is replaced by the new tag, and this tail element is moved to the head of the list. In such a case, cache miss counters for caches with the same cache set size and any values of associativity are increased.

This is illustrated for a 4-way set-associative cache in Figure 4. If the tag comparison results in a hit with the second element in the linked list, then this indicates that a cache hit would occur in the 4-way set-associative cache and the 2-way set-associative cache. A cache miss would occur in the direct mapped cache, hence it is not necessary to continue the search until the tail element. The second element is then moved to be the head element to conform with the Least Recently Used (LRU) replacement policy.

The procedure continues by traversing down the trees to find the linked list corresponding to the larger cache set size. The procedure for tag comparison and updating the associativity replacement policy is then repeated. An example is shown in Figure 2, once the cache estimation for the level $\text{cache.size} = 2$ is completed, the algorithm will look at the least significant bit of the current tag part of the address which will make up the most significant bit of the cache address for the $\text{cache.size} = 4$ tree level to determine whether to traverse the left or right branch. In Figure 2, the dotted line indicates the traversing path given the address ‘1010’.

The algorithm then continues by using the replicated forest data.
structures for simulating the different cache line sizes. Different parts of the current address are used for locating the appropriate tree data structure.

3.2 Efficiency of The Methodology

With the implementation described above, we can limit the search space for all the different cache configurations and reduce the time taken to estimate cache misses by increasing the space requirements. Each linked list entry is used to store the tag address (32 bits), a valid bit (1 bit), and a pointer to the next element in the linked list (32 bits). In total, each linked list entry needs to keep 65 bits of data. Each node in the tree needs to store pointers to the head element of the linked list (32 bits), a pointer to the left branch (32 bits), and a pointer to the right branch (32 bits); giving a total of 96 bits per node.

For each cache way, a linked list entry needs to be kept, this gives a total size of $\text{size}_{\text{list}} = A \times 65\text{bits}$. The number of nodes created is dependent on the number of cache sets, cache line size, and cache size range. The number of nodes is calculated by $\text{node}_{\text{size}} = (2^{\text{max}} - 2^{\text{min}}) \times (\log_2(L) + 1) \times 96 \text{ bits}$.

Space complexity is calculated by summing all the space needed for each node and each linked list entry. In terms of space, our data structure is optimal for storing all the necessary parameters with exception of the redundancy of the content of the linked lists. However, the space requirement is fully manageable by standard desktop computers. For the work described in this paper, we simulated cache sizes ranging from 512 bytes up to 2 MB bytes, cache associativity of 1 up to 32, and cache line size of 8 bytes up to 256 bytes. In total, we have simulated 268 cache configurations requiring 9.3 megabytes. This reasonable redundancy simplifies the maintenance of the data structure and the associated algorithms.

4. System Energy and Performance Model

To facilitate the design space exploration steps, we created crude performance and energy models for the system. The model of the embedded system architecture consisted of a processor with an instruction cache, a data cache, and embedded DRAM as main memory. The data cache uses a write-through strategy. The system architecture consisted of a processor with an instruction and data cache, and embedded DRAM as main memory.

The equation for calculating the system’s total execution time is

$$\text{Exec}_{\text{time}} = I\text{cache}_{\text{access}} \times I\text{cache}_{\text{access time}} +$$

$$I\text{cache}_{\text{miss}} \times DRAM_{\text{access time}} +$$

$$I\text{cache}_{\text{miss}} \times I\text{cache}_{\text{linesize}} \times \frac{1}{\text{DRAM}_{\text{bandwidth}}} +$$

$$D\text{cache}_{\text{access}} \times D\text{cache}_{\text{access time}} +$$

$$D\text{cache}_{\text{miss}} \times DRAM_{\text{access time}} +$$

$$D\text{cache}_{\text{miss}} \times D\text{cache}_{\text{linesize}} \times \frac{1}{\text{DRAM}_{\text{bandwidth}}} \tag{1}$$

where,

- $I\text{cache}_{\text{access}}$ and $D\text{cache}_{\text{access}}$ is the total number of memory accesses to the instruction and data cache, respectively.

- $I\text{cache}_{\text{access time}}$ and $D\text{cache}_{\text{access time}}$ is the access time of the instruction and data cache, respectively.

Energy equation of the system is given by the following equation:

$$\text{Energy}_{\text{total}} = \text{Exec}_{\text{time}} \times CPU_{\text{power}} +$$

$$I\text{cache}_{\text{access}} \times I\text{cache}_{\text{access energy}} +$$

$$D\text{cache}_{\text{access}} \times D\text{cache}_{\text{access energy}} +$$

$$I\text{cache}_{\text{miss}} \times I\text{cache}_{\text{linesize}} \times I\text{cache}_{\text{energy}} +$$

$$D\text{cache}_{\text{miss}} \times D\text{cache}_{\text{linesize}} \times D\text{cache}_{\text{energy}} +$$

$$(\text{DRAM}_{\text{access time}} + I\text{cache}_{\text{linesize}} \times \frac{1}{\text{DRAM}_{\text{bandwidth}}}) +$$

$$(\text{DRAM}_{\text{access time}} + D\text{cache}_{\text{linesize}} \times \frac{1}{\text{DRAM}_{\text{bandwidth}}}) \tag{2}$$

where,

- $CPU_{\text{power}}$ is the total processor power excluding the instruction and data cache power.

- $I\text{cache}_{\text{access energy}}$ and $D\text{cache}_{\text{access energy}}$ is the instruction cache and data cache access energy, respectively.

- $D\text{cache}_{\text{energy}}$ is the active power consumed by the DRAM.

There exist seven components in the energy equation 2. The first term $\text{Exec}_{\text{time}} \times CPU_{\text{power}}$ calculates the processor energy given that execution time takes $\text{Exec}_{\text{time}}$ amount of time. The second and third
Table 2: System Specification

<table>
<thead>
<tr>
<th>Processor Energy</th>
<th>Embedded DRAM</th>
<th>Energy</th>
<th>Latency</th>
<th>Bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td>168mW @ 100MHz</td>
<td>@100MHz</td>
<td>19.5mW</td>
<td>19.5 ns</td>
<td>50MB/sec</td>
</tr>
</tbody>
</table>

5. Experimental Procedure and Results

We compiled and simulated programs from Mediabench [15] with SimpleScalar/PISA 3.0d [16]. Program traces were generated by SimpleScalar and fed into both Dinero IV [7] and our estimation tool. Our results were completely consistent with the ones produced by Dinero IV. Our estimation tool is written in C and compiled with GNU/GCC version 3.4.3 build 20050227 with -O1 optimization. Simulations were performed on a dual Opteron64 2GHz machine with 2GB of memory. We simulated several different cache configurations, with cache sizes ranging from 512 Bytes up to 2M bytes, cache associativity ranging from 1 up to 32, and cache block sizes ranging from 8 bytes per cache line up to 256 bytes per cache line.

Table 1 shows the execution time comparison of executing Dinero IV multiple times with different cache configurations and execution of our estimation tool once. Column 1 in Table 1 shows the application name, column 2 shows the trace size of the benchmark, column 3 shows the total time taken for executing Dinero IV multiple times, column 4 shows the execution time of our estimation tool, and column 5 shows the ratio of time savings of our tool when compared to executing Dinero IV multiple times. The trace size shown in Column 2 in Table 1 only shows the size of the instruction memory access trace. From column 5 in Table 1, it can be observed that, on average, our tool is approximately 45 times faster than Dinero IV. Plotting the total execution time versus the size of the trace in Figure 8 shows that as the trace size grows exponentially, our methodology shows a linear increase in total execution time required while Dinero IV shows an exponential increase in the time required.

5.1 Result Analysis

To analyze the effect of cache miss rates on system’s performance and energy consumption, we utilized cache models from CACTI [17] for the cache access time and cache access energy. Processor energy is taken from [2]. The main memory model is taken from the embedded DRAM described in [18]. The processor and memory specification is described in Table 2. System total execution time and its energy consumption is calculated using equation 1 and equation 2, respectively.

In Figure 9, we plot the number of cache misses versus the total energy consumption for different cache configurations. The plot for cache misses versus execution time for g721enc application was shown in Figure 1. The three plots in Figure 9 show the same plot with different coloring to pick out the effect of differing cache line sizes, cache associativity, and cache sizes on the energy consumption. Figure 9(b) highlights the effect of different cache line size on the total energy consumption. Figure 9(a) highlights the effect of changing associativity on the total energy consumption. Figure 9(c) highlights the effect of varying cache sizes on the total energy consumption. Due to space constraints, we are unable to show the cache miss versus total execution time graphs with different coloring to highlight the effect of cache associativity, cache line size, and cache size on the total execution time.

5.2 Model Validation

We validate the energy model of the processor by comparing with output results of Wattch [19]. The mediabench applications were executed in Wattch and the total energy results is plotted against the total cache miss number. Figure 10 shows the total energy versus the total cache miss number for g721enc application with energy figures obtained from Wattch output.

Comparing the energy versus cache miss number graphs in Figure 10 and in Figure 9(c) show that energy results from Wattch and from equation 2 display similar patterns. As cache size gets larger, the cache miss number decreases and the energy consumption decreases; but when the cache size reaches a certain size, the energy consumption starts to increase due to compulsory misses.

It should be noted that simulation time for the 268 cache configurations with Wattch took 2.5 days. The energy values obtained from Wattch simulation has a unit of Watts.cycle and the energy values should not be compared directly against the energy values obtained from Equation 2. The energy graphs obtained using the Equation 2 and from Wattch is not an exact copy of each other. This error is due to several reasons, such as, the different processor parameters of the two processors and the inaccuracy of the simplescalar model (Wattch is built on top of SimpleScalar) for reporting cache misses. In addition, it is also known that processor energy calculation using processor model derived from simplescalar is inaccurate; for example, simplescalar modeled the issue queue, reorder buffer, and the physical register file as a unified structure called Register Update Unit (RUU), unlike in real implementations where the number of entries and the number of ports in all these components are quite disparate [20].

We also performed simulation with Wattch for the remaining Mediabench benchmarks and obtained similar energy graph results in comparison to energy graph obtain from using Equation 2. Due to space constraints, we are unable to include the energy graphs obtained with Wattch for all other benchmarks.

5.3 Design Space Exploration

Looking at the Pareto optimal points in the three plots (Figure 9), it can be concluded that for g721enc application a 16K bytes direct-mapped cache with line size of 16 bytes would be the best cache configuration in terms of lowest energy consumption. For design space exploration purposes, the best cache configuration based on performance or energy consumption can then be chosen from the performance plots and the energy plots. Best choices of cache configurations for the Mediabench benchmark is shown in Table 3.

For g721enc application, it can be seen in Table 3 that for best performance a 16K bytes direct-mapped cache with line size of 256...
bytes should be used. This indicates that the lowest energy consuming cache configurations does not translate to the fastest execution time.

6. Conclusions
In this paper, we presented a cache selection method for configurable processors. This method uses a cache simulation procedure to perform fast and accurate simulation of multiple cache configurations simultaneously using a single reading of a program trace. Our method is 45 times faster compared to existing methods of cache simulation. Fast and accurate cache miss calculations allow rapid design space exploration of optimal cache parameters for desired performance and/or energy consumption.

7. References