Frequent Loop Detection Using Efficient Non-Intrusive On-Chip Hardware

Ann Gordon-Ross and Frank Vahid*
Department of Computer Science and Engineering, University of California, Riverside
{ann/vahid}@cs.ucr.edu, http://www.cs.ucr.edu/~vahid
*Also with the Center for Embedded Computer Systems at UC Irvine

ABSTRACT
Dynamic software optimization methods are becoming increasingly popular for improving software performance and power. The first step in dynamic optimization consists of detecting frequently executed code, or "critical regions." Previous critical region detectors have been targeted to desktop processors. We introduce a critical region detector targeted to embedded processors, with the unique features of being very size and power efficient, and being completely non-intrusive to the software's execution – features needed in timing-sensitive embedded systems. Our detector not only finds the critical regions, but also determines their relative frequencies, a potentially important feature for selecting among alternative dynamic optimization methods. Our detector uses a tiny cache coupled with a small amount of logic. We provide results of extensive explorations across seventeen embedded system benchmarks. We show that highly accurate results can be achieved with only a 0.02% power overhead and acceptable size overhead. Our detector is currently being used as part of a dynamic hardware/software partitioning approach, but is applicable to a wide-variety of situations.

Categories and Subject Descriptors
B.3.2 [**Memory Structures**]: Design Styles – *Cache memories.*

General Terms: Design.

Keywords: Frequent value profiling, runtime profiling, on-chip profiling, hardware profiling, frequent loop detection, hot spot detection, dynamic optimization.

1. INTRODUCTION
Dynamic software optimization methods are becoming increasingly popular for improving software performance and power. The main reason for this trend is that dynamic optimizations have several important advantages over static approaches. Dynamic optimizations allow for a system to be optimized based on runtime behavior and values, which may be hard to determine using static methods or costly simulations, and which also may change during runtime. Furthermore, dynamic optimizations require no designer intervention and are applied transparently during runtime, meaning there is no disruption to standard software tool flows.

CASES'03, Oct. 30 – Nov. 1, 2003, San Jose, California, USA.

Recent dynamic optimization research has introduced dynamic hardware/software partitioning [26]. During execution of an application, an on-chip profiling method detects critical regions of code for hardware implementation. An on-chip tool transparently re-implements those regions on an on-chip FPGA. Subsequent executions of the application execute the critical regions of code in the FPGA, speeding up those regions by a factor of 10 or more, resulting in good overall speedups of the application.

Researchers have explored many other dynamic optimization approaches. For instance, Dynamo performs dynamic software optimizations on the most frequently executed regions of code [3]. The ProfileMe approach [8] specializes subroutines for common inputs and determines by runtime profiling which configuration to call for the best performance. Pettis and Hansen [24] improve performance by re-mapping frequently executed regions of code to non-interfering cache locations. Other approaches reduce high power memory accesses through instruction compression [10][13] or by locking instructions into a special low-power cache [4][9]. Dynamic binary translation methods store translation results from frequent code regions to improve performance as well as power [16]. Value profiling [6] determines runtime invariant variables for constant propagation and code specialization for optimized performance, or even for reduced energy.

For dynamic optimizations to be most effective, optimizations are typically applied to the most frequently executed regions of code. In embedded system applications, much of the execution time is spent in a small amount of code. Figure 1 shows the percentage of execution time spent in the corresponding percentage of code size for a large selection of MediaBench benchmarks [17]. Approximately 90% of the execution time is spent in only 10% of the code, obeying the well-known 90-10 rule. This phenomena was demonstrated for an even wider set of applications in [27]. We will refer to the 10% of code as *critical regions* of code. Detecting the critical regions of a program during run-time is an important part of any dynamic optimization approach.

Previous profiling methods are mostly targeted for a desktop computing environment, incurring runtime overhead that can be unacceptable in an embedded environment, especially for real-time embedded systems with very tight timing constraints. The most common techniques generally insert additional instructions into the binary to record profiling information or interrupt the processor at regular intervals to sample register values.

To overcome problems associated with earlier profiling methods, embedded system designers have previously relied on logic analyzers to non-intrusively profile their system. However with current systems-on-a-chip (SOCs), designers can no longer connect a logic analyzer to internal signals.

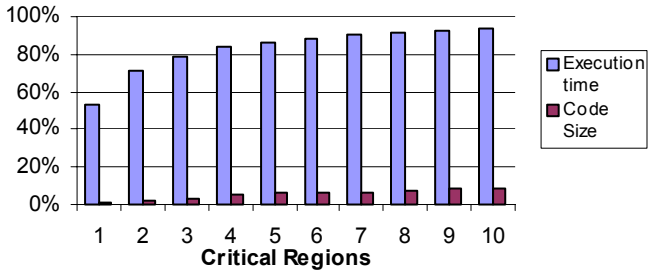


Figure 1: Average percentage of execution time spent in corresponding percentage of code size for the top N most critical code regions, for five MediaBench benchmarks.

To assist in internal signal monitoring, SOCs typically come with a means of reading internal registers via external pins utilizing the JTAG standard [12]. However, the processor must be interrupted to read the internal register values and transfer them to external pins, incurring runtime overhead and potentially altering execution behavior. This method is typically used for testing and debugging and not system profiling. Fortunately the increase in transistor capacity has also enabled on-chip profiling environments. Recent methods have been introduced that use specialized on-chip logic to profile executing applications [22][31].

In this paper, we present a new on-chip profiler that determines critical regions for use in dynamic optimizations. The profiler improves upon previous approaches by being non-intrusive, small, and low power, and also by providing information on the percentage of execution time spent in each region – information useful for guiding on-chip dynamic optimization decisions. The profiling methodology described in this paper has been shown to be very effective as the profiling step of the dynamic hardware/software partitioning approach presented in [26].

2. RELATED WORK

The most common methods for runtime profiling are software based. One such method is code instrumentation [7][11], wherein code is added to a program to count the execution frequencies of subroutines, loops or even blocks. While popular in desktop systems, instrumentation imposes program and data memory overhead and performance overhead – overheads not acceptable in many tightly constrained embedded systems. Furthermore, instrumentation may pollute instruction and data caches, and may cause register spills, resulting in very different timing behavior. Instrumentation also requires special compilers or binary instrumentation tools.

Another software-based profiling method is sampling. At certain intervals, the microprocessor is interrupted and register values are sampled [1][8], resulting in a statistical profile. Sampling reduces code and data overhead, and the sampling rate can be reduced to minimize performance overhead at the expense of accuracy. However, interrupting is intrusive and can cause problems in real-time systems. Furthermore, care must also be taken to avoid undesirable correlations between the sampling rate and the program's task periods, which could lead to aliasing problems. A method similar to interrupt-based sampling assumes a multitask environment where an additional task performs profiling in place of an interrupt [33]. However, this method has the same disadvantages as the interrupt based approach.

Another approach to profiling uses simulation. This approach uses an instruction set simulator to run the application and keep track of profiling information. Whereas a simulation-based approach can give accurate profiling information if a realistic input stimulus is available, complex external environments may be difficult, if not impossible, to model accurately – setting up an accurate simulation often takes longer than designing the application itself. Furthermore, simulation of entire systems can be extremely slow, especially for SOCs, with hours or days of simulation time correlating to only seconds of real execution time.

Many processors today come with hardware event counters that count various hardware events, such as cache misses, pipeline stalls and branch mispredictions [30][32]. Though non-intrusive, event counters do not by themselves detect critical regions of code – sampling must be used to read the counters at given intervals, thus again introducing performance overhead.

Recently, hardware-based non-intrusive profiling methods have been introduced. One method [22] utilizes a cache to determine critical regions, or "hot spots". Branch addresses and their execution frequencies are stored in a cache-like structure. Frequent branches are determined when branch frequencies reach a defined threshold value. Further analysis of branch frequencies is done to determine collections of branches that form hot spots in the code. However, this method does not focus on power efficiency and also does not store relative frequencies.

Another hardware-based methodology [29] proposes dynamic loop detection for control speculation in multithreaded processors. This method uses a stack to monitor the currently executing loops, with the innermost nested loop stored at the top of the stack and all remaining loops stored according to nesting order. When execution leaves a loop, information about loop behavior is stored into two fully-associative tables. Whereas the methodology presented may be modified to provide the loop information we require, the design was not intended for an embedded environment where power and area must be considered during the design of the profiler.

Yang and Gupta [31] proposed a very simple profiling method with low power embedded systems in mind. However, this profiling method was intended for data profiling, not code profiling. The method monitors data cache accesses and stores data values in a fully-associative table along with a small counter (2-3 bits). Each use of the data value causes the counter to be incremented. Upon counter saturation, the saturated data value is swapped with the data value in the location directly above the saturated data value in the table, effectively sorting the table, leaving the more frequent values near the top. The frequent value table is small, simple, low power and non-intrusive. However, we found that the swapping method is not accurate for code profiling, which we will elaborate on in Section 3.2.

3. FREQUENT LOOP DETECTION ARCHITECTURE

3.1 Problem Overview and Motivation

Our studies of the Powerstone [21] and MediaBench [17] benchmark suites show that about 85% of the critical regions of code are small inner loops (or near-inner loops) with the remaining 15% of the critical regions being subroutines with no inner loops.

Since 85% of the critical regions can be determined by simply finding the most frequently executed inner loops, we translate the critical code region detection problem to that of detecting frequent loops. However, in the case of benchmarks containing critical regions in the form of subroutines with no inner loops, the frequent loop detection methodology described here may be easily adapted to identify subroutines as well as loops.

A loop in an application is typically denoted by the last instruction being a short backwards branch (sbb) that jumps to the first instruction of the loop [9][18]. The sbb instruction is not a special instruction; rather an sbb is any jump instruction with a small negative offset. We examined the output of several popular C and C++ compilers using standard optimizations, and found that they indeed generate code using sbb's. In fact, we found no inner loops that were not formed using sbb's in the seventeen benchmarks we examined. However, unstructured assembly code generated by hand, or certain compiler optimizations, could result in loops with different structures. We leave frequent loop detection in these situations as future work.

In addition to detecting the most frequent loops, we also want to know those loops' percentage contribution to total execution time. Knowing the percentage contribution is important for optimization decisions. For example, suppose application X has the following loop execution breakdown: loop A 80%, loop B 5%, and loop C 5%, and application Y has the following loop execution breakdown: loop A 25%, loop B 25%, and loop C 25%. If just the order of frequent loops is known and optimizations are to be done only on the single most frequent loop, application X would yield optimizations on 80% of the execution time and application Y would yield optimizations on only 25% of the execution time. If the execution frequencies are known along with the loop ordering, optimizations on application Y can be done on the top three loops yielding optimizations on 75% of the execution time. Furthermore, with knowledge that application X's A loop takes 80% of execution time, we might perform more aggressive optimizations – such a frequent loop might be a candidate for partitioning to hardware, for example. Certain optimizations may only be applied when certain percentage thresholds are met.

We have imposed several operational requirements for our frequent loop detector: non-intrusion, low power, and small area. Non-intrusion is important for real-time systems where changes in execution behavior could significantly affect the performance of the system. Additionally, non-intrusion minimizes the impact on current tool chains, avoiding special compilers or binary modification tools. Minimal impact is important in commercial environments where significant capital may already be invested in a development environment. Minimizing power is important in low-power embedded systems, such as battery-operated systems or systems with limited cooling capabilities. Small area is also important, but is becoming less significant given the large transistor capacities of recent and future chips [15]. Another concern is that of accuracy, but our loop detector does not require exact results – instead, just reasonable accuracy is acceptable.

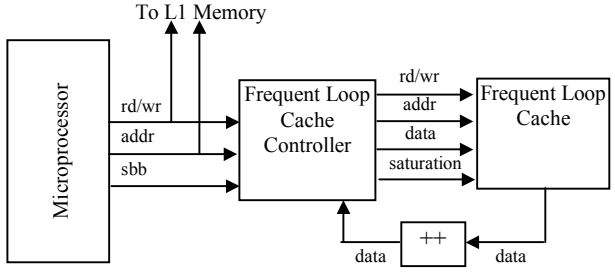


Figure 2: Frequent loop detection architecture.

3.2 Methods Considered

We initially considered many methods for determining frequent loops. We first attempted to satisfy only our first requirement of detecting the ordering of the most frequent loops by modifying the frequent data value detector design by Yang and Gupta [31]. We adapted the design so that sbb addresses would be counted instead of data values. However, we found that the frequent loops were not ordered correctly at the top of the table. We determined that the reason for the inaccurate results was because swapping of items occurs whenever an item's small counter saturates, even though the item further up in the table may have had a much higher frequency. The frequent data value method is concerned with detecting the top set of values and is not concerned with their actual ordering. We explored larger counter fields, but then the counter saturations did not happen frequently enough to allow swapping to order the frequent loops in the table.

Next we tried using a fully-associative memory to store the frequent loop addresses and their frequencies. The sbb address would be used as the tag and the tag's associated data would be incremented upon a hit. However, a fully-associative memory raised many questions such as the tradeoff between a large enough memory to give accurate results and the power consumption of that memory, as well as finding an efficient replacement policy when the memory becomes full.

We also looked into using a hash table to store frequent loops and their associated frequencies. Sbb addresses would be hashed using a subset of the address bits. By using a simple hash function and not doing too much probing, the hash table is a reasonable solution. The hashing and match detection would have to be hardware based to be non-intrusive. Incorporating hashing and match detection in hardware began to lead to a design that looked very much like a cache, which ultimately led us to the cache-based approach described in the following section.

3.3 Cache-Based Architecture

Our loop detection architecture can be seen in Figure 2. The *frequent loop cache* is a simple cache used to store frequency counts and is indexed into using sbb instruction addresses. The cache has an added feature, to be described later, that will shift every data value right by one, which is achieved by asserting the *saturation* signal to the cache. The *frequent loop cache controller* orchestrates updates to the frequent loop cache. An incrementor is also included to increment the frequency count. An additional signal, *sbb*, is required from the microprocessor, similar to that implemented in Motorola's M*CORE microprocessor [25], and that signal is asserted whenever an sbb is taken. Alternatively, if the sbb signal is not available, the cache controller could determine when an sbb is taken by replicating a small portion of the instruction decode logic.

The frequent loop cache controller handles the operation of the frequent loop cache. When the sbb signal is asserted, a read of the frequent loop cache is done using the sbb address as the index. If the

result is a hit, the frequency is read from the cache, incremented, and written back in the next cycle. If the result is a compulsory miss, the instruction is added to the cache with a frequency data value of one. If there is a conflict miss, the new address replaces the old address in the cache.

On a conflict miss, replacing the old address in the cache with the new address could cause inaccurate results, especially if two frequent loops map to the same location in the cache. One solution is to add associativity to the cache. Associativity will allow for multiple frequent loops to map to the same set without conflict. If conflicts still occur, the replacement policy used will replace the least frequent value in the set with the new incoming sbb. Whereas associativity may alleviate cache contention, situations may occur where the most frequent loops are continually replaced in the cache – a situation known as thrashing. A victim buffer may be added to the architecture to deal with cache contentions that are not solved by associativity. However, in the benchmarks we studied, a victim buffer was not necessary to achieve accurate results.

When an increment results in a frequency counter saturating, all frequency counts in the cache are divided by two using a simple right shift. The right shift operation is implemented as a special feature of the cache architecture. Such division keeps the frequency ratios reasonably accurate. While the right shifting operation can be quite power expensive, we will show that the infrequency of saturations makes the power consumed by the right shift operation insignificant with regards to the increase in average power consumption of the system.

4. EXPERIMENTS

We performed extensive experiments to determine the best size, associativity, and frequency count field width of our cache architecture. We used benchmarks from both the Powerstone [21] benchmark suite running on a 32-bit MIPS instruction set simulator, and the MediaBench [17] benchmark suite running on SimpleScalar [5]. The benchmarks are listed in Table 1.

To model power consumption of the cache memory itself, we used the Artisan memory compiler [2]. We modeled the additional logic and functionality in synthesizable VHDL using the Synopsys Design Compiler [28]. Both tools used UMC 0.18-micron CMOS technology running at 250 MHz at 1.8 V.

To determine the accuracy of each possible cache configuration, we wrote a trace simulator for the cache architecture in C++. The simulator reads in an instruction trace file for each benchmark and simulates each possible cache configuration, outputting a list of loop addresses and frequencies for each configuration.

We simulated 336 different cache configurations for each benchmark. We tested cache sizes of 16, 32, and 64 entries with direct-mapped, 2-, 4-, and 8-way associativities, and we varied the frequency counter field width from 4 to 32 bits. We determined the accuracy of the results by calculating the average difference between the actual loop execution time percentage and the calculated loop execution time percentage. For each cache configuration, we use the following formula to compute the averaged sum of differences (SOD) for the ten most frequently executed loops:

Table 1: Benchmark descriptions

Benchmark	Size of assembly in bytes	Description
adpcm	7,648	Voice Encoding
blit	4,180	Graphics Application
compress	7,480	Data Compression Program
crc	4,248	Cyclic Redundancy Check
des	6,124	Data Encryption Standard
engine	4,440	Engine Controller
epic*	154,016	Image Compression
fir	4,232	FIR Filtering
g3fax	4,384	Group Three Fax Decode
g721*	95,024	Voice Compression
jpeg	5,968	JPEG Compression
jpeg decode*	355,072	JPEG Compression
mpeg decode*	197,328	MPEG Compression
rawcaudio*	199,920	Voice Encoding
summin	4,144	Handwriting Recognition
ucbqsort	4,848	U.C.B Quick Sort
v42	6,396	Modem Encoding/Decoding

**MediaBench*

$$\frac{\sum_{i=1}^{10} \left| \%exec_{actual\,i} - \%exec_{predicted\,i} \right|^{1/2}}{10}$$

$\%exec_{actual}$ is the actual percent of execution time of a loop and $\%exec_{predicted}$ is the predicted percent of execution time output by our simulator for the same loop for a given cache configuration. The actual and predicted execution times are both in decimal representation. The result of the SOD formula gives a value between 0 and 1, with 0 being perfect accuracy, meaning no difference between the actual and predicted execution percentages. To further penalize differences between actual and predicted execution times, the difference between the two is raised to the ½ power. Raising the difference to the ½ power may at first seem counter intuitive, however keep in mind that the percentages are in decimal form and we wish to keep the value between 0 and 1.

Originally, we computed the average SOD for all loops. However, for benchmarks with a large number of loops, we found the SOD did not accurately represent the ability of the approach to calculate execution percentage of the most frequent loops.

Figure 1 shows that the first eight frequent loops comprise over 90% of the execution time while the remaining infrequent loops (possibly hundreds) share 10% of the execution time. If a critical loop detector does not identify the frequency of an infrequent loop correctly, the difference between the actual percentage of execution time and the predicted percentage of execution time will be very small. Since we are only interested in predicting the frequent loops, taking the averaged SOD for all loops can be misleading in benchmarks with many infrequent loops. The reason that the averaged SOD is misleading for benchmarks with many infrequent loops is because the slight difference in mispredictions of many infrequent loop execution times may dominate over the greater difference in mispredictions of frequent loop execution times. For

better analysis of our frequent loop detector, we will only consider the top ten most frequent loops in our average SOD calculations.

The average SOD results over all benchmarks in each benchmark suite can be seen in Figure 3. The x-axis shows the cache configuration, giving the number of ways, followed by the cache size in number of entries, followed by the frequency width in bits. For brevity, only frequency widths of 8, 12, 16, 24, and 32 bits are listed. The y-axis shows one minus the SOD so that a perfect accuracy will result in a value of 1.

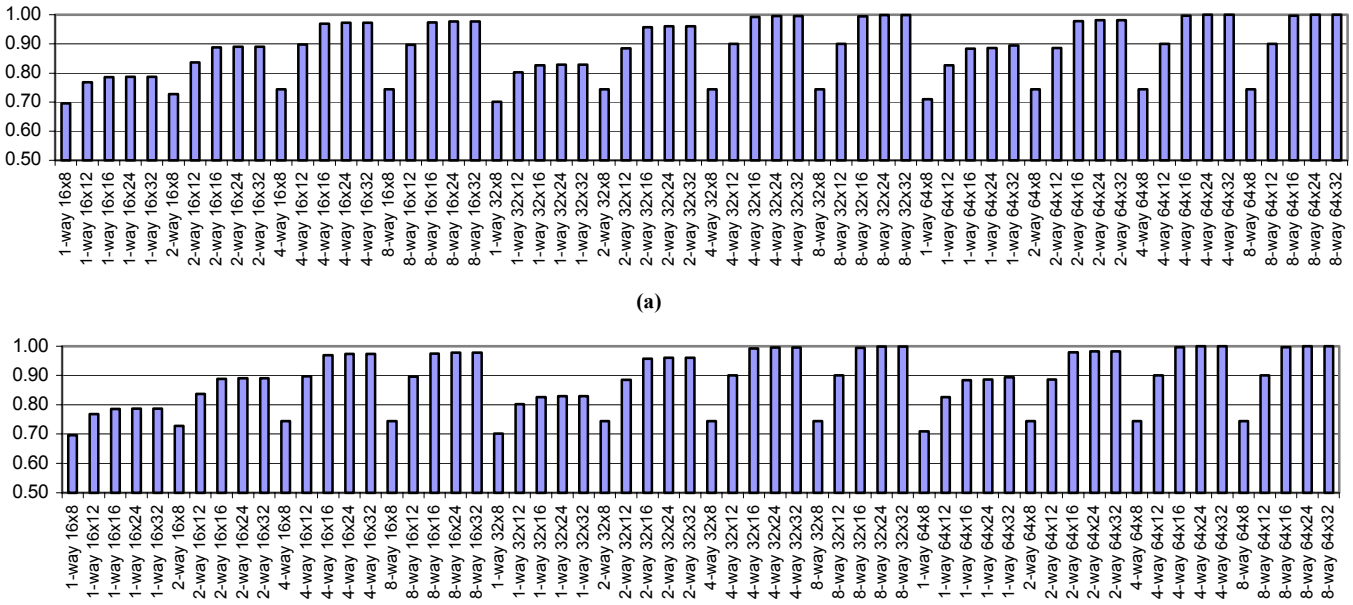
As we do not require that the results be 100% correct (90% or so is likely acceptable), we see that a good cache for both benchmark suites can be very small. By varying the frequency counter width, we are able to determine the smallest possible cache necessary to give good results, because each cache entry only contains one counter. The best cache configuration for Powerstone is a 2-way 16-entry cache with a frequency width of 16 bits, and the best cache configuration for MediaBench is a 2-way 32-entry cache with a frequency width of 24 bits. Overall, we conclude that the best overall cache configuration is a 2-way 32-entry cache with a frequency width of 24 bits. We will refer to this cache configuration as the best cache configuration. The best cache configuration is the smallest cache size that gives good results for both benchmarks suites. The 2-way/32-entry/24-bit cache yields accuracies near 95% and 90% for Powerstone and MediaBench benchmarks suites, respectively.

Figure 3 also shows that the Powerstone benchmarks tend to perform better with smaller cache configurations than does MediaBench. Thus, larger examples could require a larger cache. However, we point out that the rate of increase of the necessary cache size is low. A 16-entry cache (good for Powerstone) captures on average only 1.2% of the instructions for each Powerstone benchmark, while a 32-entry cache (good for MediaBench) captures on average only 0.13% of the instructions for each MediaBench benchmark. For even larger examples, the cache size may need to be increased, but the cache size increase is much less than the program size increase.

We now consider the power overhead of the frequent loop detector. We consider the MIPS32 4Kp microprocessor core [23], a small, low power embedded processor with a cache, having an area of 1.7 mm^2. The average power consumption for the 4Kp running at 240 MHz in 0.18-micron technology is 528 mW. The frequent loop detection hardware with the best cache configuration consumes 142 mW for each frequent loop cache read and increment, and consumes 156 mW for each frequent loop cache write, averaged over both benchmark suites. However, since only sbb instructions cause updates to the frequent loop cache, cache updates only occur an average of 4.25% of the time across all benchmarks. One saturation operation consumes 20.7 mW of power, and saturations occur only 0.000051% of the time for the best cache configuration. Thus, the resulting increase in average power consumption of the total system with the frequent loop detector is only 2.4%.

The frequent loop cache controller, incrementor and additional control/steering logic consists of 1400 gates, or an area of 0.012 mm^2. Additionally, the cache has an area of 0.167 mm^2 including saturation logic. The resulting area overhead is 10.5% compared to the reported size of the MIPS 4Kp [23]. Area actually varies greatly depending on technology libraries, foundry, etc., and thus we expect that actual area overheads would be much smaller (numbers for our cache are pessimistic, while reported microprocessor areas are likely optimistic). Nevertheless, area is becoming less constrained in nanoscale technologies.

The power consumed by the frequent loop cache can further be reduced using known methods to decrease cache power consumption, such as phased lookup or pseudo set-associative caching. Phased lookup accesses the tag arrays first, and then only accesses the hit data way. Pseudo set-associative lookup [14] essentially accesses one way (tag and data) first, and only accesses the other way upon a miss. Each technique reduces dynamic power



(a)

(b)

Figure 3: Sum of differences results for (a) Powerstone and (b) MediaBench. The x-axis shows the cache configuration with the number of ways followed by the cache size and frequency width in bits

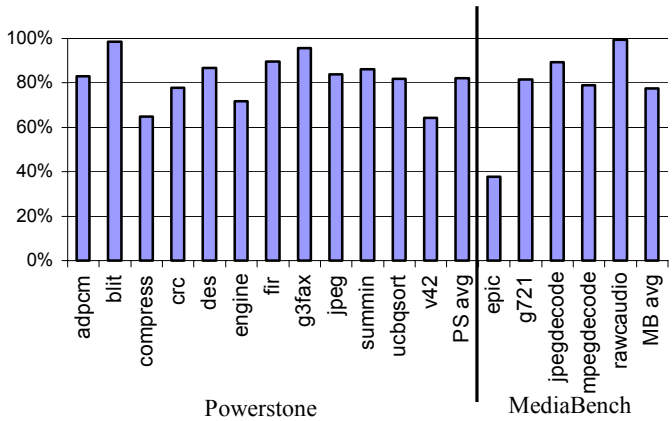


Figure 3: Percent reduction in cache updates due to the coalescing of short backwards branch increments for Powerstone and MediaBench benchmarks.

by 25% to 50%, at the expense of multi-cycle lookups – not a problem in our case since sbb's do not occur every cycle. Thus, we can easily reduce our 2.4% system power overhead to something closer to 1.5%.

5. REDUCING POWER OVERHEAD VIA FREQUENCY UPDATE COALESCING

5.1 Coalescing Methodology

The previously described method gives very good results with little power overhead, but we can further reduce power with no loss in accuracy. Frequently executed loops tend to iterate many times, causing the same sbb frequency value to be incremented in the cache many times in a row. Therefore, we can coalesce successive increments into one addition. For example, if a frequent loop executes 300 times in a row, the 300 cache frequency increments can be coalesced into one cache update with the addition of 300 to the frequency value.

We determined the potential for cache update reductions. For each benchmark, we processed the execution trace files and coalesced all of the sbb instructions. The results in Figure 4 shows average cache update reductions near 80% for both benchmark suites.

By only coalescing consecutive sbb increments, nested loops may in some cases not benefit from coalescing, with the worst case being a very highly iterated outer loop with an inner loop that iterates only a few number of times, thus alternating the sbb addresses. Coalescing could be extended to allow sbb addresses to be coalesced with, for instance, any of the last N sbb addresses seen, where N could be 2, 3, etc. We processed the trace files again allowing for sbb addresses to be coalesced with different ranges of previously seen sbb addresses. However, we found that extended coalescing did not improve significantly on the already 80% savings achieved by consecutive sbb address coalescing. Thus, we decided to extend our frequent loop cache architecture to have the ability to coalesce only consecutive sbb increments.

5.2 Coalescing Architecture

To implement the coalescing architecture, we added only a small amount of hardware to the original frequent loop detection

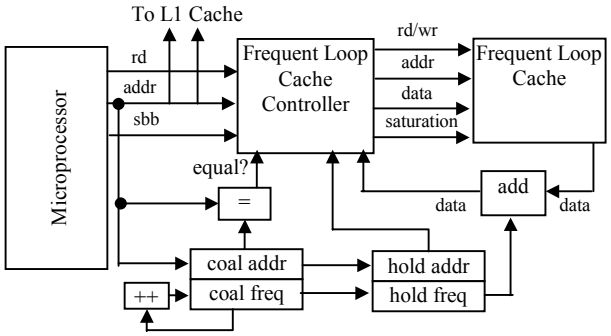


Figure 5: Frequent loop detection architecture with backwards branch coalescing. Additionally, registers and arithmetic units have load and enable signals respectively.

architecture. The new frequent loop detection architecture with coalescing hardware is shown in Figure 5. To implement coalescing, we added two sets of registers: the coalescing registers (*coal addr* and *coal freq*) and the holding registers (*hold addr* and *hold freq*). We also added an incrementor to implement the coalescing that is done in the coalescing registers, a comparator to see if the current sbb address matches the previous sbb address, and a small amount of steering logic. We replaced the incrementor, which was connected to the frequent loop cache in the previous design, by an adder to perform variable sized additions to the frequency values in the cache. We also modified the *frequent loop cache controller* to drive the new hardware.

The coalescing hardware operates as follows. For each taken sbb, the current address is compared with the coalescing address register. If there is a match, the coalescing freq is incremented to tally this execution. If there is no match, the address and frequency in the coalescing registers are moved into the holding registers and the new sbb address is written to the coalescing register. The data in the frequent loop cache is then updated to reflect the values in the holding registers. Cache hits and misses are handled the same way as they were in the frequent loop detector without coalescing. Furthermore, saturations in the coalescing frequency register cause a right shift by one in both the coalescing register and all values currently stored in the frequency cache.

5.3 Coalescing Results

The experimental setup for the frequent loop cache detector with coalescing is the same as the setup for the design without coalescing. We modeled the additional coalescing hardware in synthesizable VHDL, resulting in an area overhead of approximately 2300 gates or an area of 0.020 mm^2. Control/steering logic, registers and arithmetic units are included in the gate count. The area of the cache itself remains the same as the frequent loop detector without coalescing. The cache with coalescing hardware represents an 11% increase to the area overhead of the MIPS 4Kp.

A power savings of 98.9% is achieved by coalescing one sbb instead of doing one cache update, with the lowest and highest savings being 97.5% and 99.7% respectively, depending on frequency size. Thus, the power consumed by coalescing is insignificant compared to a cache update.

To see total system power savings by using coalescing, we must determine the new power overhead related to total system power using the MIPS system described in Section 4 and the best cache

configuration. With the addition of the coalescing hardware, cache updates now only occur on average 0.91% of the time across all benchmarks. Coalescing one sbb consumes only 2.3 mW of power and coalescing occurs on average 3.3% of the time across all benchmarks. The resulting increase in average power consumption of the total system with the frequent loop detector with coalescing is now reduced to a mere 0.53%, i.e., less than 1% power overhead.

Along with the benefit of reduced power consumption, the coalescing hardware still preserves the fidelity of the results. Since no instruction executions are lost, only coalesced, the accuracy of the SOD results for each cache configuration is identical to those achieved with the frequent loop detector without coalescing.

6. SAMPLING FOR FURTHER REDUCED POWER OVERHEAD

In conjunction with coalescing, sbb instruction sampling can also be used to further reduce the power overhead, at the expense of some accuracy. Instead of tallying every sbb instruction executed, only sbbs that occur at fixed sampling intervals will be included in the frequency counts. This method does not require interrupting of the microprocessor, as previous sampling methods required. The frequent loop cache controller will only tally sbbs that occur on the sampling interval – such sampling is easily implemented using a small (e.g., 6-bit) counter.

To see the impact of sbb instruction sampling on the accuracy of the results, we simulated the best cache configuration for sampling intervals of 1, 5, 25 and 50 sbb instructions. The results can be seen in Figure 6 for each benchmark. For all Powerstone benchmarks (except *jpeg*), the average trend is the degradation of accuracy as the sampling rate gets larger. On average for the Powerstone benchmark suite, 5% of accuracy is lost when going from a sampling interval of 1 to 50. However, for the MediaBench benchmark suite, the average trend is for the accuracy of the results to *improve* by approximately 2% with a sampling rate of 50. The reason for the improvement in accuracy is because sampling causes a decrease in saturations. For MediaBench, saturations cause more information to be lost than sampling does.

At a sampling rate of 50, the cache updates and coalesces decrease even further to rates of 0.03% and 0.06% respectively with no saturations. Coalescing plus sampling (at a rate of 50) reduces the average system power overhead to a mere 0.02% (0.05% without coalescing).

7. EXAMPLE USE: WARP PROCESSING

The frequent loop detector described in this paper can be used in a variety of situations. The detector has been successfully incorporated into a novel prototype system-on-a-chip architecture performing what is presently known as a warp processor [19][20][26], also developed at the University of California, Riverside. The architecture consists of microprocessors coupled with field-programmable gate arrays (FPGAs), along with a single dynamic partitioning module that itself contains a lean microprocessor. The dynamic partitioning module monitors the software executing on each regular microprocessor (one microprocessor at a time), detects the critical software kernels, and automatically remaps those kernels to an FPGA coprocessor. Such remapping typically speeds up a kernel by a factor of 10 or more.

The warp processor architecture designers successfully incorporated our frequent loop detector into their architecture and use that detector to find the critical kernels. Overall application speedups obtained by remapping of those kernels from a 75 MHz ARM to FPGA are presently between 2 and 3, but substantially faster speedups are projected as more aggressive transformations (e.g., loop unrolling, loop pipelining) and more efficient FPGA fabrics are developed. For highly parallel examples, estimated speedups of greater than 10 can be achieved when performing aggressive optimizations.

The frequent loop detector is presently being incorporated into another project's architecture (a joint project between the University of California, Riverside and the University of California, Irvine), in which a prototype platform SOC is used to help speed up desktop CAD algorithms. The detector will be incorporated onto the platform, with the results fed back to desktop tools used to obtain profiles and to guide hardware/software partitioning, cache configuration, and other algorithms.

8. CONCLUSIONS

We introduced a small, power efficient architecture for accurately and non-intrusively detecting the most frequent loops of an executing program, while accurately providing the relative

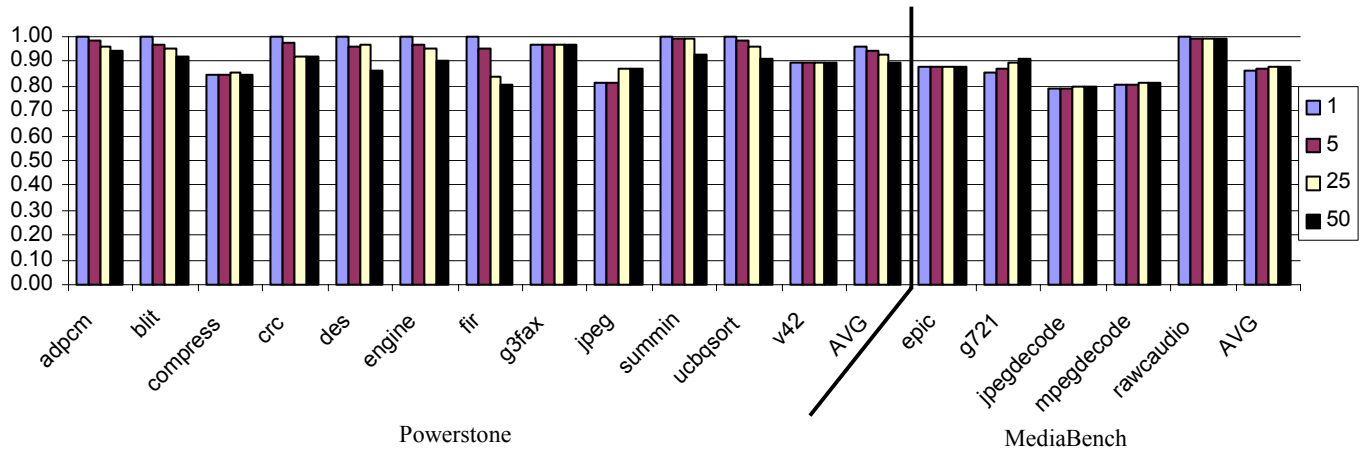


Figure 6: Sum of differences compared to the perfect loop frequencies for the best cache, a 2-way set-associative 32 entry cache with a frequency width of 24 bits, for Powerstone and MediaBench benchmarks with short backwards branch sampling intervals of 1, 5, 25 and 50 instructions.

frequencies of those loops. We displayed the effectiveness of the architecture using numerous benchmarks. The architecture uses a 2-way set-associative 32-entry cache with each entry storing a 24-bit frequency counter. We show that power overhead of our loop detector is only 1-2% compared to a 32-bit embedded processor, and is easily reducible to well below 0.1% using simple coalescing and sampling methods. Future work involves extending the design to find critical subroutines as well as critical loops.

9. ACKNOWLEDGEMENTS

This work was supported in part by the U.S. National Science Foundation (grants CCR-0203829 and CCR-9876006) and a Department of Education GAANN fellowship.

10. REFERENCES

[1] Anderson, J., Berc, L.M., Dean, J., Ghemawat, S., Henzinger, M.R., Leung, S.T.A., Sites, R.L., Vandevoorde, M.T., Waldspurger, C.A., Weihl, W.E. Continuous profiling: where have all the cycles gone? 16th ACM Symp. of Operating Systems Design, 1997.

[2] Artisan, http://www.artisan.com.

[3] Bala, V., Duesterwald, E., Banerjia. Dynamo: a transparent dynamic optimization system. Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implemenation, 2000.

[4] Bellas, N., et al. Energy and performance improvements in microprocessor design using a loop cache. ICCD, pp. 378-383, 1999.

[5] Burger, D., Austin, T., Bennet, S. Evaluating future microprocessors: the simplescalar toolset. University of Wisconsin-Madison. Computer Science Department Tech. Report CS-TR-1308, July 2000.

[6] Calder, B., Feller, P., Eustace, A. Value profiling. MICRO pp. 259-267, 1997.

[7] Cmelik, R., SpixTools – introduction and user's manual, Sun Microsystems Laboratories, Inc. Technical Report SMLI TR 93-6, 2/93.

[8] Dean, J., Hicks, J., Waldspurger, C.A., Weihl, W.E., Chrysos, G. ProfileMe: Hardware support for instruction level profiling on out-of-order processors, MICRO 1997.

[9] Gordon-Ross, A., Cotterell, S., Vahid, F. Exploiting fixed programs in embedded systems: a loop cache example. IEEE Computer Architecture Letters, Vol 1, January 2002.

[10] Govindarajan, S.C., Ramaswamy, G., Mehendale, M. Area and power reduction of embedded DSP systems using instruction compression and re-configurable encoding. International Conference on Computer Aided Design, 2001.

[11] Grahm, S.L., Kessler, P.B., McKusick, M.K. Gprof: a call graph execution profiler. SIGPLAN Symp. on Compiler Construction, 1982.

[12] IEEE, IEEE 1149.1 Standard Test Access Port and Boundary-Scan Architecture, http://standards .ieee.org, 2001.

[13] Ishihara, Y., Yasuura, H. A power reduction technique with object code merging for application specific embedded processors. Design Automation and Test in Europe, March 2000.

[14] Hennessy, J.L. and Patterson, D.A. Computer architecture: a quantitative approach. Morgan Kaufmann, 1990.

[15] Kiefendorff, K.. Transistor Budgets Go Ballistic. Microprocessor Report, Volume 12, Number 10, August 1998, pp. 34-43.

[16] Klaiber, A. The technology behind crusoe processors. Transmeta Technical Brief. January 2000.

[17] Lee, C., Potkonjak, M., Mangione-Smith, W.H. MediaBench: a tool for evaluating and synthesizing multimedia and communication systems. Proc 30th Annual International Symposium on Microarchitecture, Dec 1997.

[18] Lee, L.H., Moyer, B., Arends, J. Instruction fetch energy reduction using loop caches for embedded applications with small tight loops. International Symposium On Low Power Electronics and Design, 1999.

[19] Lysecky, R, Vahid, F. A codesigned on-chip logic minimizer. First IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2003.

[20] Lysecky, R., Vahid, F. On-chip logic minimization. Proceedings of the 40th ACM/IEEE Conference on Design Automation (DAC), 2003.

[21] Malik, A., Moyer, W., Cermak, D. A low power unified cache architecture providing power and performance flexibility. ISLPED, 2000.

[22] Merten, M.C., Trick, A. R., George, C.N., Gyllenhaal, J., Hwu, W.W. A hardware-driven profiling scheme for identifying program hot spots to support runtime optimization. ISCA 1999.

[23] MIPS Technologies, http://www.mips.com/content/Products/Cores/32-BitCores/MIPS324KFamily/ProductCatalog/P_MIPS324KFamily/productBrief

[24] Pettis, K., Hansen, R.C. Profile guided code positioning. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), June 1990.

[25] Scott, J., Lee, L.H., Chin, A., Arends, J., Moyer, W. Designing the M*CORE M3 CPU architecture. IEEE International Conference on Computer Design (ICCD), 1999.

[26] Stitt, G., Lysecky, R., Vahid, F. Dyanmic hardware/software partitioning: a first approach. Proceedings of the 40th ACM/IEEE Conference on Design Automation (DAC), 2003.

[27] Suresh, D.C., Najjar, W.A., Vahid, F., Villarreal, J.R., Stitt, G. Profiling tools for hardware/software partitioning of embedded applications. Languages, Compilers and Tools for Embedded Systems (LCTES), 2003, pp. 189-198.

[28] Synopsys Inc., http://www.synopsys.com.

[29] Tubella, J., Gonzalez, A. Control speculation in multithreaded processors through dynamic loop detection. In Proceedings of the Fourth International Symposium On High Performance Computer Architecture (HPCA), 1998.

[30] Vtune Environment, Intel Corp., http://developer.intel.com/vtune

[31] Yang, J., Gupta, Rajiv. Energy efficient frequent value data cache design. MICRO 2002.

[32] Zagha, M., Larson, B., Turner, S., Itzkowitz, M. Performance analysis using the MIPS R10000 performance counters. Supercomputing, Nov. 1996.

[33] Zhang, X., et al. System support for automatic profiling and optimizations. Proceedings of the 16th Symposium on Operating System Principles, 1997.