Frequent Loop Detection Using Efficient Nonintrusive On-Chip Hardware

Ann Gordon-Ross, *Student Member*, *IEEE*, and Frank Vahid, *Member*, *IEEE*

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." Most 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 nonintrusive 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-like structure coupled with a small amount of logic. We provide results of extensive explorations across 19 embedded system benchmarks. We show that highly accurate results can be achieved with only a 0.02 percent power overhead, acceptable size overhead, and zero runtime 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.

Index Terms—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. A generic microprocessor with dynamic optimization capabilities can also tune itself to any application running on the microprocessor. Furthermore, dynamic optimizations require no intervention from the application designer and are applied transparently during runtime, meaning there is no disruption to standard software tool flows.

Researchers have explored many dynamic optimization techniques. For instance, Warp Processing [28], [29], [34] determines the most frequently executed regions of code and dynamically moves those regions of code to hardware reducing energy consumption and increasing performance. Dynamo [6] performs dynamic software optimizations on the most frequently executed regions of code. Other approaches reduce high-power memory accesses through instruction compression [18], [22] or by locking instructions into a special low-power cache [10], [17]. Dynamic binary translation methods store translation results from frequently executed code regions to improve performance as well as power [16], [25].

- *The authors are with the Department of Computer Science, University of California, Riverside, Riverside, CA 92521.*
E-mail: {ann, vahid}@cs.ucr.edu.

Manuscript received 15 Apr. 2004; revised 8 Mar. 2005; accepted 5 Apr. 2005; published online 16 Aug. 2005.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TCSI-0128-0404.

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. Fig. 1 shows the percentage of execution time spent in the corresponding percentage of code size for all of the benchmarks studied. Approximately 90 percent of the execution time is spent in only 10 percent of the code, obeying the well-known 90-10 rule for embedded applications. We will refer to the code comprising the 90 percent of execution time as the *critical regions* of code. Suresh et al. [35] studied this rule further and determined that this phenomena was demonstrated for an even wider set of embedded applications, with roughly 90 percent of the execution time spent in the first two to four most frequently executed loops.

Previous profiling methods are mostly targeted for a desktop computing environment and applications where significant information must be gathered about the execution of an application such as information necessary for determining the cause of pipeline stalls. Gathering large quantities of information can incur significant runtime overhead that can be unacceptable in an embedded environment, especially for timing-sensitive embedded systems with very tight timing constraints. However, timing-sensitive systems, such as a video decoder or a fingerprint scanner, can benefit significantly from runtime dynamic optimizations, but may not tolerate even minimal runtime overhead, possibly causing the system to miss hard deadlines. To minimize overhead, very low overhead hardware-based profiling methods exist; however, most of them do not provide the percentage of execution time spent in critical regions of code. Additionally, previous methods do not report area and power overhead of the profiling

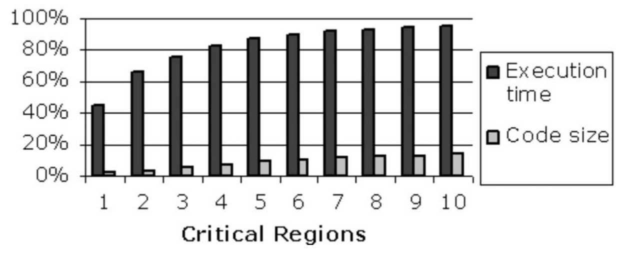


Fig. 1. Average percentage of execution time spent in corresponding percentage of code size for the top N most critical code regions, for the benchmarks studied.

hardware, information that is critical to the design of small, low-power embedded systems.

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 completely nonintrusive, 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 step of the dynamic hardware/software partitioning approach presented in [34] effectively utilizes the profiling methodology described in this paper.

2 Related Work

The most common methods for runtime profiling are software based. One such method is code instrumentation [13], [19], wherein code is added to a program to count the execution frequencies of subroutines, loops, or even blocks.

Edge profiling [32] determines the transition frequency between basic blocks by incrementing a counter each time a branch instruction is executed. Edge profiling determines the most frequently executed paths through an application by following the highest frequency edges. Whereas edge profiling is typically used for link-time optimizations, the concept of edge profiling could be adapted to a runtime mechanism by instrumenting the application. However, incrementing a counter on each basic block transition introduces significant overhead. Ball and Larus [7] use instrumentation for a low-overhead path profiling method to determine how many times each acyclic path in a routine is executed. Path profiling improves upon the overhead incurred by edge profiling by reducing the overhead from an average of 31 percent for edge profiling down to an average of 16 percent for path profiling. The reduction in overhead is achieved through the reduction of counter updates. In edge profiling, each instrumented edge has a counter that is incremented each time an edge executes, while path profiling only increments a counter after the completion of execution through a unique path through the code.

While popular in desktop systems, instrumentation imposes program and data memory overhead and performance overhead—overheads not acceptable in many tightly constrained embedded systems. Overhead of instrumented code is commonly in the range of 30-1,000 percent [3], [8], [12] with reports of an overhead as high as 10,000 percent (100 times slower) [12].

Arnold and Ryder [4] present a method for reducing the overhead incurred by code instrumentation. Code duplication and instrumentation sampling is used to reduce overhead. Code duplication allows for two copies of the code to be available—an instrumented copy of the code and a noninstrumented copy of the code. A sampling method determines how often the instrumented code is executed. This method can achieve profiles 93-98 percent accurate with an average total overhead of only 3 percent.

Even with the possibility of drastically reduced instrumentation overheads, even the slightest overhead may be unacceptable for rigid timing-sensitive systems with hard deadlines. 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], [14], resulting in a statistical profile. Anderson et al. [2] describe the Digital Continuous Profiling Infrastructure (DCPI) tools for transparent, low-overhead profiling of complete systems. The DCPI tools are the profiling step used in ProfileMe [14]. The DCPI tools identify events and the instructions that cause these events by profiling on a per instruction bases at a specified sampling rate. Profiling is done with specialized hardware to record information about the instruction as the instruction moves through the pipeline. Upon instruction completion, software reads the sampling information into a database. The database may then be analyzed to determine which instructions may benefit from various optimization methods. Typically, statistical sampling gathers much more information about an application than what would be needed for determining the critical regions of code. To further reduce the overhead of gathering information during the interrupt, only information related to determining critical regions could be recorded.

While sampling reduces code and data overhead over instrumentation, the sampling rate can be reduced to minimize performance overhead at the expense of accuracy, and only information pertinent to determining critical regions would be gathered, interrupting is still intrusive and can cause problems in timing-sensitive systems. Even minimal performance overhead can cause hard deadlines in critical timing-sensitive systems to be missed. 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, although randomized sampling alleviates this problem. A method similar to interrupt-based sampling assumes a multitask environment where an additional task performs profiling in place of an interrupt [42]. However, this method has the same disadvantages as the interrupt-based approach.

Another type of statistical sampling is program phase identification [9], [15]. As a program executes, the program passes through many different phases, each of which can have vastly different performance. Hardware counters with little to no runtime overhead record information about the executing program. An interrupt then triggers either the hardware or software to transfer the hardware counter values to auxiliary data storage and analyzes the recorded information to determine the phases of the application. Optimizations can then be applied to the most critical phases. Most phase identification methods induce runtime overhead and those that do not induce runtime overhead do not discuss the impacts of phase identification on area and power consumption.

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 [21], [39], [41]. Though nonintrusive, 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.

To overcome problems associated with earlier profiling methods, embedded system designers have previously relied on logic analyzers to nonintrusively profile their system. However, with current systems-on-a-chip (SOCs), designers can no longer connect a logic analyzer to internal signals. To assist in internal signal monitoring, SOCs typically come with a means of reading internal registers via external pins utilizing the JTAG standard [20]. 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.

One hardware-based nonintrusive profiling method [30] 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 or area overhead and also does not store relative frequencies.

Another hardware-based methodology proposes dynamic loop detection for control speculation in multithreaded processors [37]. 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 [40] 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 nonintrusive. 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 [33] benchmark suite shows that a large percentage of time is spent in small loops [38]. Averaged across all benchmarks, 66 percent of the total execution time spent in loops only is spent in loops of size 256 instructions or less. However, we also noticed that 77 percent of this time is spent in loops of size 32 instructions or less. The loops of size 32 instructions or less account for 51 percent of the total execution time spent in loops only. For the Powerstone and MediaBench [27] benchmarks studied, we observed that about 85 percent of the critical regions of code were these small inner loops (or near-inner loops—typically no more than two to four levels of nesting) with the remaining 15 percent of the critical regions being subroutines with no inner loops that were called from a frequently executed loop. Since 85 percent 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. Detecting only frequent loops does not limit the usefulness of our tool in applications where many critical regions are in the form of subroutines with no loops. Through observation, we concluded that we could assume that any dynamically executed subroutine called by a critical loop was indeed critical regions of code as well. Furthermore, the incident of frequent subroutines with no inner loops can be reduced by inlining functions. However, in the cases where function inlining is not available or an application has a large number of critical subroutines, 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 backward branch (sbb) that jumps to the first instruction of the loop [17], [27]. 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 sbbs. In fact, we found no inner loops that were not formed using sbbs in the 19 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 percent, loop B 5 percent, and loop C 5 percent, and application Y has the following loop execution breakdown: loop A 25 percent, loop B 25 percent, and loop C 25 percent. 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 percent of the execution time and application Y would yield optimizations on only 25 percent 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 percent of the execution time. Furthermore, with knowledge that application X's A loop takes 80 percent 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: nonintrusion, low power, and small area. Nonintrusion is important for real-time systems where changes in execution behavior could significantly affect the performance of the system. Additionally, nonintrusion 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 [24]. Another concern is that of accuracy, but our loop detector does not require exact results—instead, just reasonable accuracy is acceptable.

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 [40] to count sbb instructions instead of data values. However, the frequent loops were not ordered correctly at the top of the table. 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.

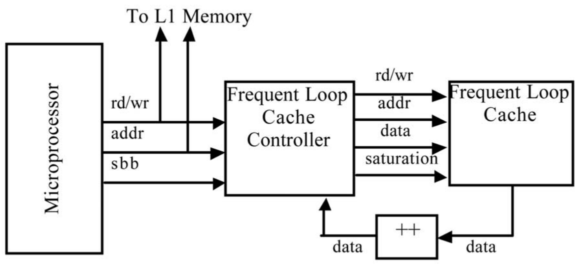


Fig. 2. Frequent loop detection architecture.

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 trade-off 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.

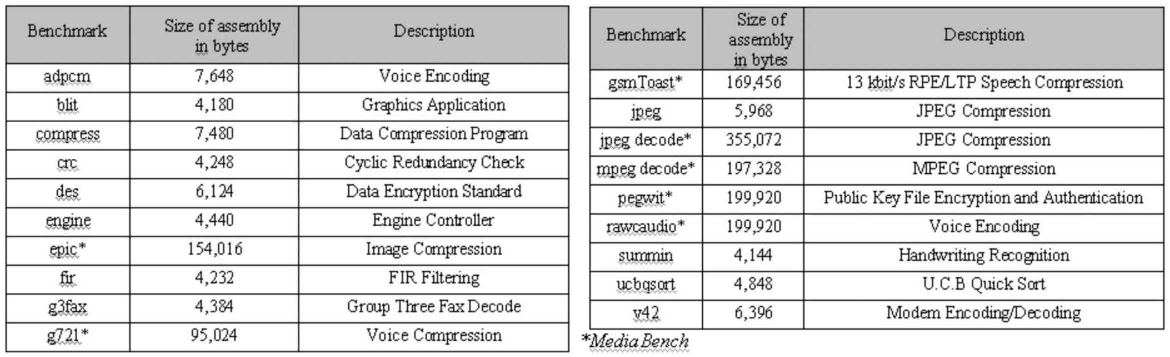
We also looked into using a hash table to store frequent loops and their associated frequencies. Sbb addresses would be hashed using a subset of the address bits. By using a simple hash function and not doing too much probing, the hash table is a reasonable solution. The hashing and match detection would have to be hardware based to be nonintrusive. 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

Fig. 2 shows our loop detection architecture. 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 [33], 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 with a frequency of one. By evicting the conflicting address, any frequency information gathered about that address is lost. To save that information, the

TABLE 1
Benchmark Description



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
gsmToast*	169,456	13 kbit/s RPE/LTP Speech Compression
jpeg	5,968	JPEG Compression
jpeg decode*	355,072	JPEG Compression
mpeg decode*	197,328	MPEG Compression
pegwit*	199,920	Public Key File Encryption and Authentication
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

address and corresponding frequency would need to be stored in an auxiliary memory structure, such as the main memory of the microprocessor. However, this method leads to increased complexity to ensure that no conflicts exist between the loop detection logic and the application while accessing main memory. Furthermore, on subsequent conflict misses, an efficient method for searching evicted addresses is required and may be difficult to do since the number of sbb instructions can be very large. To avoid these added complexities, the evicted address is simply removed from the cache.

However, 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 a 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 regard to the increase in average power consumption of the system.

4 Experiments

4.1 Setup and Evaluation Framework

We performed extensive experiments to determine the best size, associativity, and frequency count field width of our cache architecture to minimize area and power consumption. We chose benchmarks from both the Powerstone [33] benchmark suite running on a 32-bit MIPS instruction set simulator and the MediaBench [26] benchmark suite running on SimpleScalar [11]. Table 1 lists, for each benchmark, the size of the assembly code in bytes and a short description. We were unable to run the remaining MediaBench benchmarks because our experimental method requires instruction traces of the application and the remaining MediaBench benchmarks produce very large and cumbersome instruction traces.

To model power consumption of the cache memory itself, we used the Artisan memory compiler [5]. We modeled the additional logic and functionality in synthesizable VHDL using the Synopsys Design Compiler [36] to obtain power and area estimates. 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.

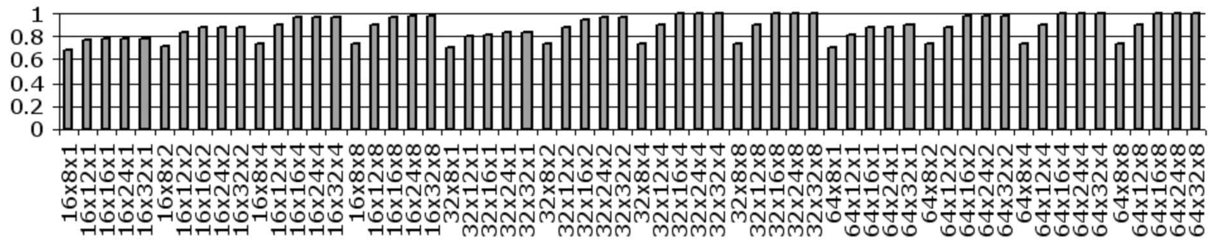
We simulated 336 different cache configurations for each benchmark. We tested cache sizes of 16, 32, and 64 entries with direct-mapped, 2, 4, and 8-way associativities, and we varied the frequency counter field width from 4 to 32 bits. We determined the accuracy of the results by calculating the average 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 squared (SOD) for the 10 most frequently executed loops:

$$\frac{\sum_{i=1}^{10} \left| \%exec_{actual_i} - \%exec_{predicted_i} \right|^{1/2}}{10}.$$

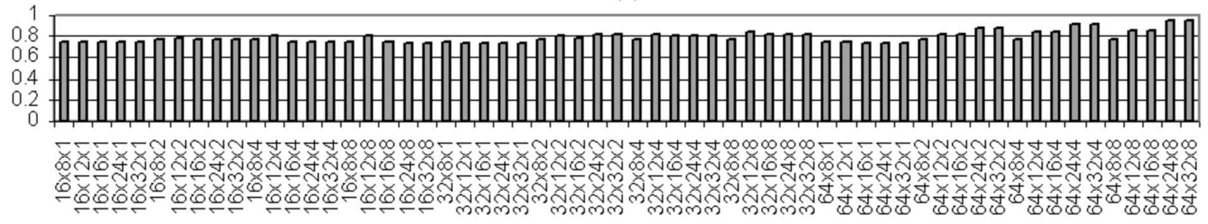
$\%\text{exec}_{\text{actual}}$ is the actual percent of execution time of a loop and $\%\text{exec}_{\text{predicted}}$ is the predicted percentage of execution time output by our simulator for the same loop for a given cache configuration. If our profiler does not predict a loop's execution time, the predicted value is set to 0 for that loop; however, this situation rarely happened and, when it did occur, the unpredicted loop typically contributed to only a very small fraction of execution time. The actual and predicted execution times are both in decimal

TABLE 2
Sum of Differences Squared (SOD) Calculation for the Top 10 Most Frequently Executed Loops for epic

	Loop 1	Loop 2	Loop 3	Loop 4	Loop 5	Loop 6	Loop 7	Loop 8	Loop 9	Loop 10	
$\%\mathbf{exec}_{\mathbf{actual}}$	.47939	.25448	.03196	.02011	.01414	.01414	.01341	.01341	.01341	.01341	
$\%\mathbf{exec}_{\mathbf{predicted}}$	.57168	.30348	.03811	.02398	0	.01686	.01599	0	0	0	
$\lvert\%\mathbf{exec}_{\mathbf{actual}}-\%\mathbf{exec}_{\mathbf{predicted}}\rvert$	.09229	.049	.00615	.00384	.01414	.00272	.00258	.01341	.01341	.01341	**SOD**
$\lvert\%\mathbf{exec}_{\mathbf{actual}}-\%\mathbf{exec}_{\mathbf{predicted}}\rvert^{1/2}$	.30379	.22136	.07842	.06197	.11891	.05215	.05079	.11580	.11580	.11580	**.189**
											18.9%



(a)



(b)

Fig. 3. Sum of differences results for (a) Powerstone and (b) Mediabench. The x-axis shows the cache configuration with cache size in number of entries, followed by the frequency width in bits, followed by the associativity.

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 1/2 power. Raising the difference to the 1/2 power may at first seem counterintuitive, however, keep in mind that the percentages are in decimal form and we wish to keep the value between 0 and 1. Table 2 shows the SOD calculation for the *epic* benchmark. The table shows that the SOD method heavily penalizes differences in frequencies. For example, the difference in percentage of execution time for loop 1 is 9.2 percent and the SOD value is 30.4 percent—significantly penalizing this small difference.

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. Fig. 1 shows that, on average, the first 10 frequent loops comprise over 90 percent of the execution time, while the remaining infrequent loops (possibly hundreds) share 10 percent 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 10 most frequent loops in our average SOD calculations.

4.2 Results

Fig. 3 shows the average SOD results over all benchmarks in each benchmark suite. The x-axis shows the cache configuration, giving the cache size in number of entries, followed by the frequency width in bits, followed by the associativity. The cache configurations are ordered first by increasing number of entries, second by increasing frequency width size, and, last, by increasing associativity. The cache configurations are ordered in this manner so that the cache size essentially increases from left to right—smaller cache configurations are toward the left and larger cache configurations are toward the right. This ordering method makes choosing the best cache configuration with the smallest size very easy to determine. 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.

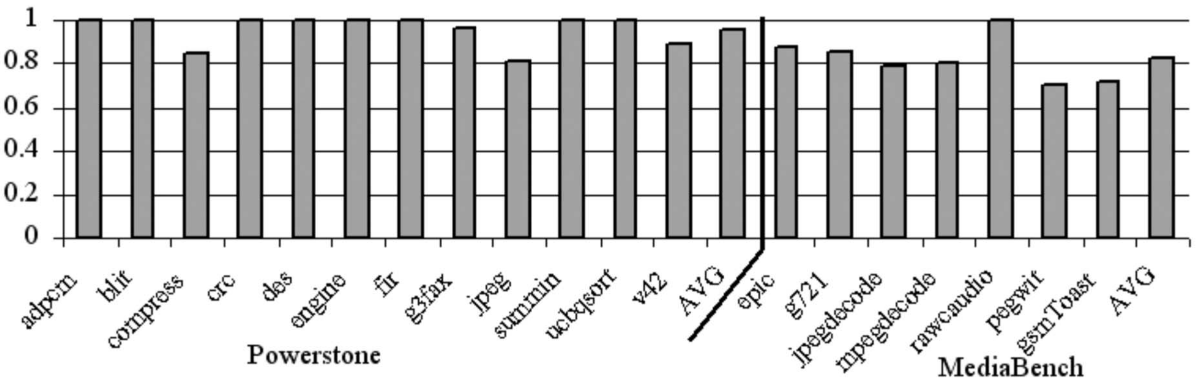


Fig. 4. Sum of differences results for each benchmark for the best cache configuration (32-entry 2-way set associative cache with a frequency field width of 24 bits).

TABLE 3
Actual (Act.) and Predicted (Pred.) Percentage Execution Times for the Top 10 Most Frequently Executed Loops for a Selection of Benchmarks for the Best Cache Configuration (2-Wy/32-Entry/24-Bit)

	Loop 1		Loop 2		Loop 3		Loop 4		Loop 5		Loop 6		Loop 7		Loop 8		Loop 9		Loop 10		1-SOD
	Act.	Pred.	Act.	Pred.	Act.	Pred.	Act.	Pred.	Act.	Pred.	Act.	Pred.	Act.	Pred.	Act.	Pred.	Act.	Pred.	Act.	Pred.	
compress	17.0%	20.5%	17.0%	20.5%	17.0%		12.9%	15.5%	11.7%	14.0	9.9%	11.9	6.6%	8.0%	5.4%	6.6%	0.8%	0.9%	0.5%	0.6%	84.6%
g3fax	47.0%	47.2%	46.1%	46.3%	6.3%	6.4%	0.3%		0.3%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	n/a	n/a	97.2%
jpeg	19.1%	24.9%	19.1%	24.9%	16.5%	21.5%	9.5%	12.4%	9.2%		5.9%		3.4%	4.4%	2.6%		2.5%	3.2%	2.4%	3.1%	81.4%
v42	42.3%	46.1%	9.6%	10.4%	8.9%	9.7%	8.9%	9.7%	8.2%		6.4%	7.0%	3.5%	3.8%	3.5%	3.8%	2.9%	3.2%	2.8%	3.0%	89.6%
epic	47.9%	57.2%	25.4%	30.3%	3.2%	3.8%	2.0%	2.4%	1.4%		1.4%	1.7%	1.3%	1.6%	1.3%		1.3%		1.3%		87.6%
g721	79.0%	93.8%	4.8%		4.0%	4.8%	4.0%	1.0%	1.6%		0.8%		0.8%		0.8%		0.8%		0.8%		85.8%
jpegdecode	29.9%	42.5%	29.7%	42.5%	6.0%		6.0%		5.5%	7.9%	4.5%		4.2%		3.8%	5.4%	1.3%		0.9%		78.9%
mpegdecode	35.3%	46.5%	35.3%	46.5%	4.4%		4.4%	5.8%	4.4%		4.4%		3.2%		1.3%		1.3%		1.2%		80.7%
rawcaudio	99.3%	99.4%	0.3%	0.3%	0.1%	0.1%	0.1%	0.1%	0.1%	0.1%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	99.7%
pegwit	32.7%	67.7%	32.6%		5.9%	12.2%	5.9%	12.2%	4.7%		3.4%		1.9%		1.9%		1.8%		1.8%	3.8%	70.9%
gsmToast	33.0%	66.9%	8.3%		4.1%		4.1%		4.1%		4.1%		4.1%	8.4%	4.1%	8.4%	4.1%		4.1%	8.4%	72.2%

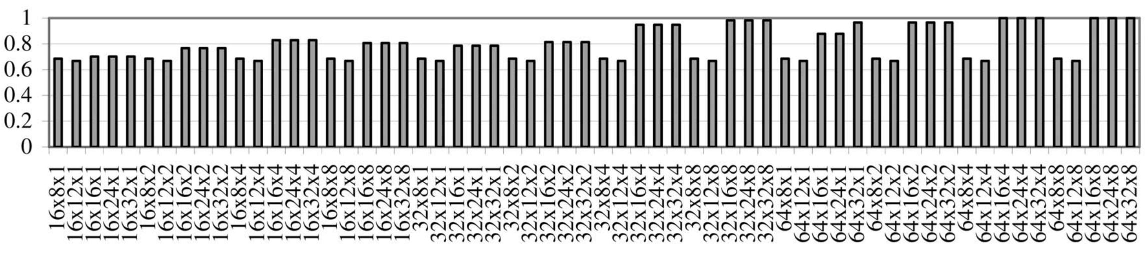
Empty cells are loops that were not predicted by the frequent loop detector.

We compared the predicted loop frequencies and the actual loop frequencies for each benchmark for each cache configuration and concluded that we do not require that the results be 100 percent correct—near 80 to 90 percent or so is likely acceptable. Results 80 to 90 percent accurate correctly identified the most critical loops in all but two benchmarks. Given this threshold, 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 percent and 85 percent for the Powerstone and MediaBench benchmarks suites, respectively.

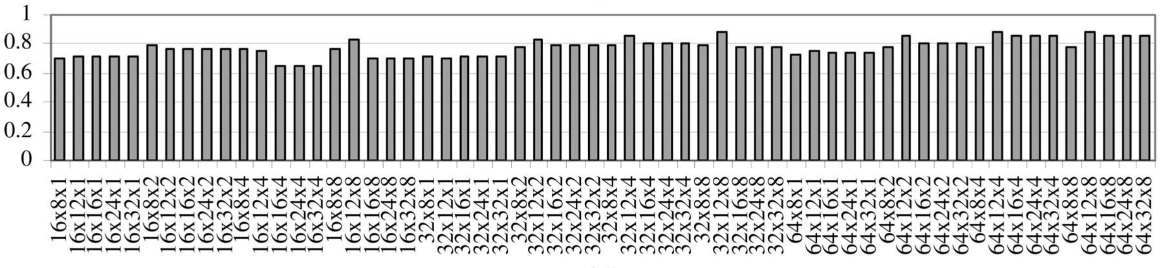
Fig. 3 also shows that the Powerstone benchmarks tend to perform better with smaller cache configurations than does MediaBench. Thus, larger examples could require a larger cache. However, we point out that the rate of increase of the necessary cache size is low. A 16-entry cache (good for Powerstone)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

Fig. 4 more closely inspects the effectiveness of the best cache configuration (32-entry, 2-way set-associative, 24-bit frequency field). Fig. 4 shows the sum of differences results for each benchmark for the best cache configuration. For 10 benchmarks, the best cache configuration produces perfect or near perfect results. For all but two benchmarks, results are at least 80 percent accurate—still very reasonable accuracy. For two benchmarks, *pegwit* and *gsmToast*, the results are only 72 percent accurate. Table 3 further explores the details of these benchmarks.

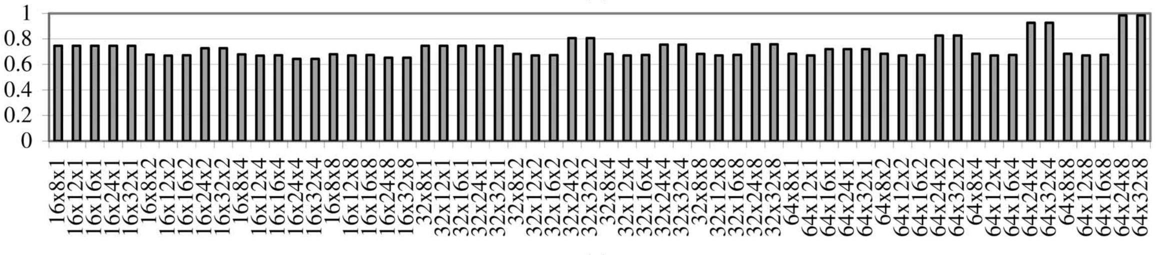
Table 3 shows the actual and predicted percentage execution times for a selection of interesting benchmarks—benchmarks with an SOD value of 100 percent are not shown. Table 3 shows that, even though some loop frequencies are not predicted accurately, the information obtained is very useful. For instance, one of the worst SOD values obtained was for the *gsmToast* benchmark. *GsmToast*



(a)



(b)



(c)

Fig. 5. Sum of difference results for (a) the Powerstone benchmark *jpeg*, (b) the MediaBench benchmark *jpegDecode*, and (c) the MediaBench benchmark *mpegDecode*. The x-axis shows the cache configuration with cache size in number of entries, followed by the frequency width in bits, followed by the associativity.

has one very frequent loop comprising 33 percent of the execution time and all other loops account for less than 9 percent of the execution time. The frequent loop detector predicted loop one to account for 67 percent of the execution time and did not even predict a value for most of the loops with less than 9 percent of the execution time. A designer applying optimizations would most likely only be interested in loops comprising at least 10 percent to 20 percent of the execution time. Even though the frequent loop detector did not predict the exact frequency of loop one, the most important information is still conveyed—the benchmark has one very critical region of code. Other benchmarks, such as *jpegdecode, g721*, and *mpegDecode*, show similar trends.

Empty cells in Table 3 indicate situations where the frequent loop detector did not predict a frequency for a particular loop. For *compress* and *pegwit*, the frequent loop detector does not identify the third and second most frequently executed loop, accounting for 17 percent and 33 percent of the execution time, due to conflicts in the cache. Increased associativity could have eliminated this anomaly, but we point out that, for all 19 benchmarks, the frequent loop detector only missed these two loops and still correctly identified the other critical loops for these applications. Additionally, the frequent loop detector never identified a noncritical loop as being a critical loop.

Fig. 5 further investigates three benchmarks with low accuracy: Powerstone *jpeg*, MediaBench *jpegdecode*, and MediaBench *mpegdecode*. The accuracy for *jpeg* increases to nearly perfect results by simply increasing to 4-way set associativity. Similarly, the accuracy for *jpegdecode* increases to 90 percent by increasing to 8-way associativity. For the third benchmark, *mpegdecode*, both the size and the associativity need to increase to 64-entry and 4-way, respectively. We observed that these benchmarks tended to have a significantly larger number of loops than the other benchmarks studied. The changes to the frequent loop cache required to produce more accurate results are needed to deal with the larger number of loops to keep track of. Increasing the size of the cache to 64 entries and the associativity to 4-way increased the accuracies for all three benchmarks to over 90 percent. Nevertheless, the 32-entry/2-way/24-bit frequency configuration works well enough across all the benchmarks, yielding average accuracy of over 90 percent across all benchmarks and no lower than 80 percent for any of the benchmarks, which is acceptable. Thus, we decided to keep the 32-entry/2-way/24-bit frequency configuration for the profiler to minimize the size and power impact of the profiler on a microprocessor system. However, a 64-entry configuration might also be reasonable. In Section 6, we will show that we can improve the accuracy of the least-accurate benchmarks using a sampling technique.

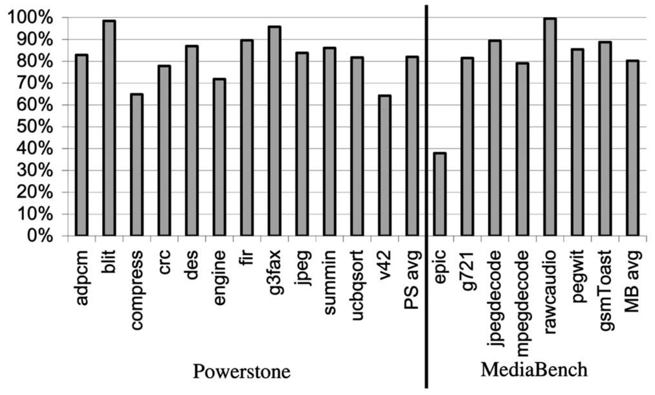


Fig. 6. Percent reduction in cache updates due to the coalescing of short backwards branch increments for Powerstone and MediaBench benchmarks.

4.3 Area and Power Overhead

We now consider the power overhead of the frequent loop detector. We consider the MIPS32 4Kp microprocessor core [31], a small, low power embedded processor with a cache, having an area of 1.4 mm^2 to 2.5 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 percent of the time across all benchmarks. One saturation operation consumes 20.7 mW of power and saturations occur only 0.000051 percent of the time for the best cache configuration. Thus, the resulting increase in the average power consumption of the total system with the frequent loop detector is only 2.4 percent.

The power consumed by the frequent loop cache can be further 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. Pseudoset-associative lookup [23] essentially accesses one way (tag and data) first and only accesses the other way upon a miss. Each technique reduces dynamic power by 25 percent to 50 percent, at the expense of multicycle lookups—not a problem in our case since sbbs do not occur every cycle. Thus, we can easily reduce our 2.4 percent system power overhead to something closer to 1.5 percent.

The frequent loop cache controller, incrementor, and additional control/steering logic consist of 1,400 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 6.68 percent to 12.8 percent compared to the reported size range of the MIPS 4Kp [31]. 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).

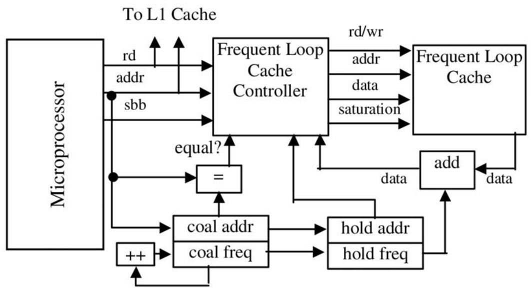


Fig. 7. Frequent loop detection architecture with backward branch coalescing. Additionally, registers and arithmetic units have load and enable signals, respectively.

Nevertheless, with transistor capacity increasing at a tremendous rate [31], area is becoming less constrained in nanoscale technologies. With excessive chip area available, it is not uncommon to see new microprocessor and SOC designs with dedicated logic for debugging, monitoring, or tuning of the system.

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 Fig. 6 show average cache update reductions near 80 percent 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 small 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 80 percent savings already 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 architecture. Fig. 7 shows the new frequent loop

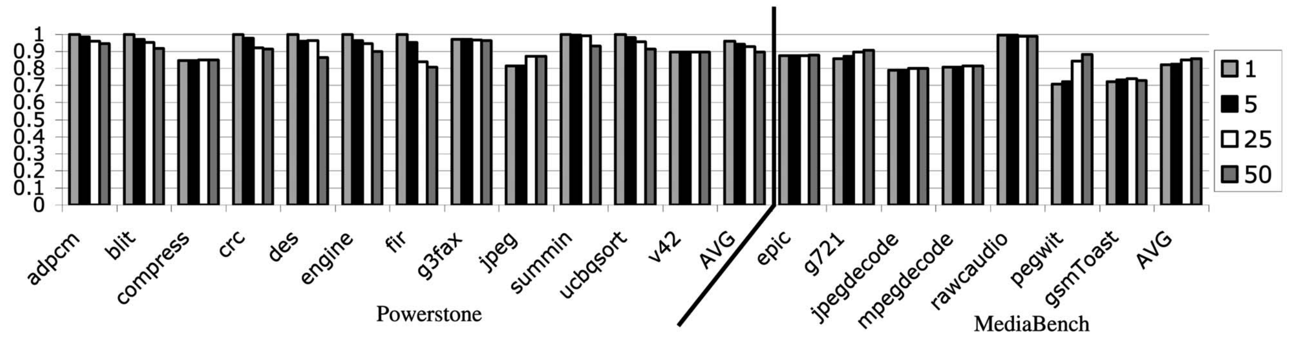


Fig. 8. Sum of differences compared to the perfect loop frequencies for the best cache, a 2-way set-associative 32 entry cache with a frequency width of 24 bits, for Powerstone and MediaBench benchmarks with short backward branch sampling intervals of 1, 5, 25, and 50 instructions.

detection architecture with coalescing hardware. 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, with 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 2,300 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 a 7.48 to 13.3 percent increase to the area overhead of the MIPS 4Kp.

A power savings of 98.9 percent is achieved by coalescing one sbb instead of doing one cache update, with the lowest and highest savings being 97.5 percent and 99.7 percent, 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 percent of the time across all benchmarks. Coalescing one sbb consumes only 2.3 mW of power and coalescing occurs, on average, 3.3 percent of the time across all benchmarks. The resulting increase in the average power consumption of the total system with the frequent loop detector with coalescing is now reduced to a mere 0.53 percent, i.e., less than 1 percent 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 interruption 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. Fig. 8 shows the results 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 percent 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 percent with a sampling rate of 50. Much of the inaccuracy is due to conflicts in the cache structure, where infrequent loops evict frequent loops. Since the MediaBench

TABLE 4
Actual (Act.) and Predicted (Pred.) Percentage Execution Times for the Top 10 Most Frequently Executed Loops for a Selection of Benchmarks for the Best Cache Configuration (2-Way/32-Entry/24-Bit) with a Sampling Rate of 1 (s1) and a Sampling Rate of 50 (s50)

	Loop 1		Loop 2		Loop 3		Loop 4		Loop 5		Loop 6		Loop 7		Loop 8		Loop 9		Loop 10		1-SOD
	Act.	Pred.	Act.	Pred.	Act.	Pred.	Act.	Pred.	Act.	Pred.	Act.	Pred.	Act.	Pred.	Act.	Pred.	Act.	Pred.	Act.	Pred.	
jpeg-s1	19.1%	24.9%	19.1%	24.9%	16.5%	21.5%	9.5%	12.4%	9.2%		5.9%		3.4%	4.4%	2.6%		2.5%	3.2%	2.4%	3.1%	81.4%
jpeg-s50	19.1%	21.4%	19.1%	21.8%	16.5%	18.6%	9.5%	11.1%	9.2%	10.2%	5.9%		3.4%	4.1%	2.6%		2.5%	2.8%	2.4%	2.7%	87.2%
g721-s1	79.0%	93.8%	4.8%		4.0%	4.8%	4.0%	1.0%	1.6%		0.8%		0.8%		0.8%		0.8%		0.8%		85.8%
g721-s50	79.0%	85.0%	4.8%	5.2%	4.0%	4.3%	4.0%		1.6%	1.8%	0.8%	0.9%	0.8%	0.9%	0.8%	0.0%	0.8%		0.8%		89.8%
pegwit-s1	32.7%	67.7%	32.6%		5.9%	12.2%	5.9%	12.2%	4.7%		3.4%		1.9%		1.9%		1.8%		1.8%	3.8%	70.9%
pegwit-s50	32.7%	36.5%	32.6%	36.6%	5.9%	6.5%	5.9%	6.5%	4.7%	5.3%	3.4%		1.9%	2.0%	1.9%	2.2%	1.8%		1.8%	2.0%	88.1%

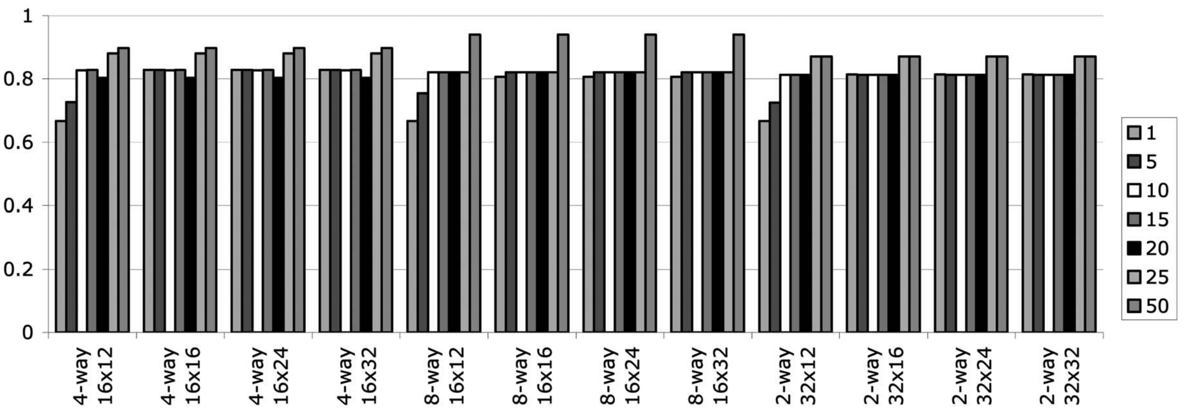


Fig. 9. Sum of differences compared to the perfect loop frequencies for a selection of frequent loop cache configurations for jpeg for sampling rates ranging from 1 to 50 sbb instructions.

benchmarks tend to be much larger than Powerstone benchmarks, MediaBench benchmarks have a much larger number of loops. For benchmarks with a smaller number of loops, there is not much contention in the cache. However, for benchmarks with a large number of loops, a large sampling rate reduces the possibility that an infrequent loop will ever be cached, thus reducing cache contention. Table 4 further investigates the benchmarks with the most improved results due to sampling. Table 4 shows the SOD calculation for each benchmark for a sampling rate of 1 (s1) and a sampling rate of 50 (s50). For every benchmark, the results show greatly improved accuracy for all loops accounting for more than 10 percent of the execution time. In the *pegwit* benchmark, the frequent loop detector without sampling was unable to detect loop two accounting for 32.6 percent of the execution time. However, with a sampling rate of 50, the frequent loop detector not only detected this loop, but also detected the loop with a reasonably accurate frequency of 36.6 percent.

Fig. 9 looks at effects of sampling more closely for the *jpeg* benchmark for a selection of interesting frequent loop cache configurations and a larger number of sampling rates. The effects of the sampling rates are different for different cache configurations. In a 4-way 16-entry cache with a frequency width of 12 bits, going from no sampling to a sampling rate of 50 increases the accuracy of the cache by 23 percent. However, for a 4-way 16-entry cache with a frequency width of 32 bits and a sampling rate of 10, the accuracy actually decreases over no sampling. These unpredictable results are due to changes in cache contention.

Even though sampling has varying degrees of effect on accuracy, we notice that the effects on the best cache configuration are quite favorable. For benchmarks with a low accuracy before sampling, i.e., *jpeg, jpegdecode, mpegdecode*, and *mpegwit*, sampling tended to increase the accuracy due to reduced contention in the cache. For the benchmarks with near perfect accuracies before sampling, sampling tended to decrease the accuracy due to loss of information, but by an acceptable amount. Overall, sampling is beneficial on average across all benchmarks because it improves the results of the less accurate benchmarks with only minimal impact on the highly accurate benchmarks.

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

Thus, to minimize the power overhead of the profiler without too much loss of accuracy overall, and even improved accuracy for some benchmarks, we might choose a sampling rate of 50.

7 Example Uses

7.1 Warp Processing

The detector has been successfully incorporated into a novel prototype system-on-a-chip architecture performing what is presently known as warp processing [28], [29], [34]. The

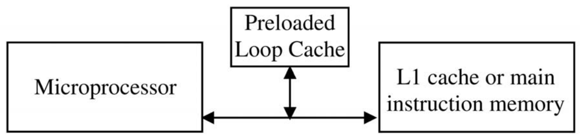


Fig. 10. Memory hierarchy role for the preloaded loop cache.

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.

The warp processor architecture designers successfully incorporated our frequent loop detector into their architecture and use that detector to find the critical kernels. Overall, an application speedup of 8 is obtainable on average by remapping the critical kernels from a 200 MHz MIPS to FPGA and 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.

7.2 Frequent Loop Caching

Another architecture focuses on caching frequently executed regions of code in a small level 0 cache near the microprocessor in embedded systems in what is called a preloaded loop cache [17]. As shown in Fig. 10, the preloaded loop cache is not simply another level of cache, but is a small table used to fetch instructions that are guaranteed not to miss in the frequent loop cache. The small size of the preloaded loop cache allows the cache to be placed very close to the microprocessor, resulting in very fast access times. The small size also allows for very low power fetches. We showed that the preloaded loop cache results in a 70 percent reduction in instruction memory accesses [17].

The preloaded loop cache incurs no performance overhead since only fetches that are guaranteed not to miss are sent to the preloaded loop cache. Previous methods perform an offline profiling step to determine the frequently executed regions of code for placement into the preloaded loop cache. The requirement of offline profiling limited the applicability of the preloaded loop cache because of the design time required executing the profiling step and the difficulty of setting up an accurate simulation environment. Furthermore, difficulty arises in a multiapplication environment because all applications must be known ahead of time and profiled with the critical region information stored on-chip for loading into the preloaded loop cache during application swap.

The frequent loop detector described in this paper significantly extends the applicability of the preloaded loop cache. The frequent loop detector allows for transparent use of the preloaded loop cache. Instead of an offline profiling step for determination of the frequent regions of code, the frequent loop detector can be included on-chip to perform the profiling step. The frequent regions of code would then be loaded into the preloaded loop cache and the preloaded loop cache would service subsequent fetches of instructions in the critical regions.

The coupling of the preloaded loop cache and the frequent loop detector also allows for easy incorporation into a multiapplication environment. The profiling step can be performed after each application swap and the critical regions can be loaded into the preloaded loop cache. To reduce profiling time, the critical regions can be stored for later lookup when execution returns to the application.

Furthermore, the preloaded loop cache could even be configured for different execution phases of an application. Applications typically have different phases of execution, each of which has different frequently executed regions of code. The number of accesses bypassing the preloaded loop cache would be monitored and, when a given threshold is reached, a phase change would be detected. At that time, the frequent loop detector would be activated to determine the new frequently executed regions of code to be loaded in to the preloaded loop cache. As with a multiapplication environment, the critical regions for each phase can be stored for later lookup when execution returns to that phase in the application.

8 Conclusions

We introduced a small, power-efficient architecture for accurately and nonintrusively detecting the most frequent loops of an executing program, while accurately providing the relative 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 the power overhead of our loop detector is only 1-2 percent, compared to a 32-bit embedded processor, and is easily reducible to well below 0.1 percent using simple coalescing and sampling methods.

Acknowledgments

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

References

[1] J. Anderson, L.M. Berc, J. Dean, S. Ghemawat, M.R. Henzinger, S.T.A. Leung, R.L. Sites, M.T. Vandevoorde, C.A. Waldspurger, and W.E. Weihl, "Continuous Profiling: Where Have All the Cycles Gone?" *Proc. 16th ACM Symp. Operating Systems Design,* 1997.

[2] J. Anderson, L. Berc, G. Chrysos, J. Dean, S. Ghemawat, J. Hicks, S.T. Leung, M. Lichtenberg, M. Vandevoorde, C.A. Waldspurger, and W.E. Weihl, "Transparent, Low-Overhead Profiling on Modern Processors," *Proc. Workshop Profile and Feedback-Directed Compilation,* Oct. 1998.

[3] G. Ammons, T. Ball, and J. Larus, "Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling," *Proc. SIGPLAN '97 Conf. Programming Language Design and Implementation,* 1997.

[4] M. Arnold and B.G. Ryder, "A Framework for Reducing the Cost of Instrumented Code," *Proc. Conf. Programming Language Design and Implementation,* 2001.

[5] Artisan, http://www.artisan.com, 2003.

[6] V. Bala, E. Duesterwald, and S.. Banerjia, "Dynamo: A Transparent Dynamic Optimization System," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implemenation,* 2000.

[7] T. Ball and J. Larus, "Efficient Path Profiling," *Proc. 29th Ann. Int'l Symp. Microarchitecture,* 1996.

[8] T. Ball and J. Larus, "Optimally Profiling and Tracing Programs," *ACM Trans. Programming Languages and Systems,* vol. 16, no. 4, pp. 1319-1360, July 1994.

[9] R.D. Barnes, E.M. Nystrom, M.C. Merten, and W.W. Hwu, "Vacuum Packing: Extracting Hardware-Detected Program Phases for Post-Link Optimization," *Proc. MICRO,* 2002.

[10] N. Bellas et al., "Energy and Performance Improvements in Microprocessor Design Using a Loop Cache," *Proc. Int'l Conf. Computer Design (ICCD),* pp. 378-383, 1999.

[11] D. Burger, T. Austin, and S. Bennet, "Evaluating Future Microprocessors: The Simplescalar Toolset," Technical Report CS-TR-1308, Computer Science Dept., Univ. of Wisconsin-Madison, July 2000.

[12] B. Calder, P. Feller, and A. Eustace, "Value Profiling and Optimization," *J. Instruction Level Parallelism,* vol. 1, Mar. 1999.

[13] R. Cmelik, "SpixTools—Introduction and User's Manual," Technical Report SMLI TR 93-6, Sun Microsystems Laboratories, Inc., Feb. 1993.

[14] J. Dean, J. Hicks, C.A. Waldspurger, W.E. Weihl, and G. Chrysos, "ProfileMe: Hardware Support for Instruction Level Profiling on Out-of-Order Processors," *Proc. 30th Int'l Symp. Microarchitecture,* 1997.

[15] A. Dhodapkar and J. Smith, "Managing Multi-Configuration Hardware via Dynamic Working Set Analysis," *Proc. 29th Ann. Int'l Symp. Computer Architecture,* 2002.

[16] K. Ebcioglu and E. Altman, "DAISY: Dynamic Compilation for 100% Architecture Compatibility," *Proc. 24th Int'l Symp. Computer Architecture,* pp. 26-37, June 1997.

[17] A. Gordon-Ross, S. Cotterell, and F. Vahid, "Exploiting Fixed Programs in Embedded Systems: A Loop Cache Example," *IEEE Computer Architecture Letters,* vol. 1, Jan. 2002.

[18] S.C. Govindarajan, G. Ramaswamy, and M. Mehendale, "Area and Power Reduction of Embedded DSP Systems Using Instruction Compression and Re-Configurable Encoding," *Proc. Int'l Conf. Computer Aided Design,* 2001.

[19] S.L. Grahm, P.B. Kessler, and M.K. McKusick, "Gprof: A Call Graph Execution Profiler," *Proc. SIGPLAN Symp. Compiler Construction,* 1982.

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

[21] Intel Corp., *Intel Itanium 2 Processor Reference Manual for Software Development and Optimization,* June 2002.

[22] Y. Ishihara and H.A. Yasuura, "A Power Reduction Technique with Object Code Merging for Application Specific Embedded Processors," *Proc. Design Automation and Test in Europe,* Mar. 2000.

[23] J.L. Hennessy and D.A. Patterson, *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 1990.

[24] K. Kiefendorff, "Transistor Budgets Go Ballistic," *Microprocessor Report,* vol. 12, no. 10, pp. 34-43, Aug. 1998.

[25] A. Klaiber, "The Technology behind Crusoe Processors," Transmeta Technical Brief, Jan. 2000.

[26] C. Lee, M. Potkonjak, and W.H. Mangione-Smith, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communication Systems," *Proc 30th Ann. Int'l Symp. Microarchitecture,* Dec. 1997.

[27] L.H. Lee, B. Moyer, and J. Arends, "Instruction Fetch Energy Reduction Using Loop Caches for Embedded Applications with Small Tight Loops," *Proc. Int'l Symp. Low Power Electronics and Design,* 1999.

[28] R. Lysecky and F. Vahid, "A Codesigned On-Chip Logic Minimizer," *Proc. First IEEE/ACM/IFIP Int'l Conf. Hardware/Software Codesign and System Synthesis (CODES+ISSS),* 2003.

[29] R. Lysecky and F. Vahid, "On-Chip Logic Minimization," *Proc. 40th ACM/IEEE Conf. Design Automation (DAC),* 2003.

[30] M.C. Merten, A.R. Trick, R.D. Barnes, E.M. Nystrom, C.N. George, J. Gyllenhaal, and W.W. Hwu, "An Architectural Framework for Run-Time Optimizations," *IEEE Trans. Computers,* vol. 50, no. 6, pp. 567-589, June 2001.

[31] MIPS Technologies, http://www.mips.com/content/Products/Cores/32-BitCores/MIPS324KFamily/ProductCatalog/P_MIPS324KFamily/productBrief, 2003.

[32] K. Pettis and R.C. Hansen, "Profile Guided Code Positioning," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI),* June 1990.

[33] J. Scott, L.H. Lee, A. Chin, J. Arends, and W. Moyer, "Designing the M*CORE M3 CPU Architecture," *Proc. IEEE Int'l Conf. Computer Design (ICCD),* 1999.

[34] G. Stitt, R. Lysecky, and F. Vahid, "Dyanmic Hardware/Software Partitioning: A First Approach," *Proc. 40th ACM/IEEE Conf. Design Automation (DAC),* 2003.

[35] D.C. Suresh, W.A. Najjar, F. Vahid, J.R. Villarreal, and G. Stitt, "Profiling Tools for Hardware/Software Partitioning of Embedded Applications," *Proc. Languages, Compilers and Tools for Embedded Systems (LCTES),* pp. 189-198, 2003.

[36] Synopsys Inc., http://www.synopsys.com, 2003.

[37] J. Tubella and A. Gonzalez, "Control Speculation in Multithreaded Processors through Dynamic Loop Detection," *Proc. Fourth Int'l Symp. High Performance Computer Architecture (HPCA),* 1998.

[38] J. Villareal, R. Lysecky, S. Cotterell, and F. Vahid, "Loop Analysis of Embedded Applications," Technical Report UCR-CSE-01-03, Univ. of California, Riverside, 2001.

[39] Vtune Environment, Intel Corp., http://developer.intel.com/vtune, 2003.

[40] J. Yang and R. Gupta, "Energy Efficient Frequent Value Data Cache Design," *Proc. MICRO,* 2002.

[41] M. Zagha, B. Larson, S. Turner, and M. Itzkowitz, "Performance Analysis Using the MIPS R10000 Performance Counters," *Proc. Supercomputing,* Nov. 1996.

[42] X. Zhang et al., "System Support for Automatic Profiling and Optimizations," *Proc. 16th Symp. Operating System Principles,* 1997.

Ann Gordon-Ross is currently a PhD student in the Department of Computer Science at the University of California, Riverside. She received the BS degree in computer science from the University of California, Riverside in 2000. Her research interests include low-power embedded-system design with an emphasis on low-power cache design. She is a student member of the IEEE and a member of the ACM.

Frank Vahid received the BS degree in computer engineering from the University of Illinois Urbana/Champaign in 1988 and the MS and PhD degrees in computer science from the University of California, Irvine, in 1990 and 1994, respectively, where he was an SRC Fellow. He is a professor of computer science and engineering at the University of California, Riverside, and a faculty member of the Center for Embedded Computer Systems at the University of California, Irvine. He has coauthored more than 100 conference and journal papers, including the best paper award from the *IEEE Transactions on VLSI* in 2000. He is author of the textbook *Digital Design* (John Wiley and Sons, 2006) and coauthor of the textbooks *Embedded System Design* (John Wiley and Sons, 2002) and *Specification and Design of Embedded Systems* (Prentice Hall, 1994). He received the Outstanding Teacher of the UCR College of Engineering award in 1997 and the College's Teaching Excellence Award in 2003. He was program and general chair for the IEEE/ACM International Symposium on System Synthesis in 1996 and 1997, respectively, and for the IEEE/ACM International Workshop on Hardware/Software Codesign in 1999 and 2000. His current research focuses on designing novel self-tuning architectures for low-power embedded and desktop systems and on creating a generation of electronic blocks that nonexperts and experts alike can utilize to build basic but useful sensor-based embedded systems. He is a member of the IEEE and the IEEE Computer Society