Non-Intrusive Dynamic Application Profiler for Detailed Loop Execution Characterization

Ajay Nair, Roman Lysecky
Department of Electrical and Computer Engineering
University of Arizona
{ajaynair, rlysecky}@ece.arizona.edu

Abstract

Application profiling – the process of monitoring an application to determine the frequency of execution within specific regions – is an essential step within the design process for many software and hardware systems. In this paper, we present an efficient innovative, non-intrusive dynamic application profiler (DAProf) capable of profiling an executing application by monitoring the application's short backwards branches and providing detailed profiling statistics for characterizing loop execution behavior. DAProf is ideally suited for hardware/software partitioning approaches in which detailed loop execution information is needed to provide accurate performance estimates. DAProf provides a profiling accuracy of greater than 90% with only an 11% area overhead compared to a small ARM9.

Categories and Subject Descriptors
C.4 **[Computer Systems Organization]** Performance of Systems – *Measurement Techniques*.

General Terms: Design, Performance.

Keywords: Profiling, nonintrusive, dynamic optimization, embedded systems.

1. INTRODUCTION

Application profiling – the process of monitoring an application to determine the frequency of execution within specific regions – is an essential step within the design process for many software and hardware systems. Profiling has long been utilized to identify the most frequently executed regions of a software application such that a developer can focus their efforts on optimizing those regions [13]. Binary translation and dynamic optimization techniques rely on dynamic profiling to determine frequently executed sequences of instructions and improve performance by either caching binary translation results or re-compiling the code sequences [2][6][10][18]. Profiling has also been utilized to create several specialized software [5][27] or hardware implementations [19], from which an application can select at runtime which to execute to improve performance or reduce power consumption. In [3][11][21], profiling is used to detect frequent loops to map to a small loop cache to reduce power consumption.

Profiling is a critical step within hardware/software partitioning approaches in which an application is partitioned into software executing on a microprocessor and one or more hardware coprocessors. Application profiling is often utilized to determine an application's frequently executed regions, or critical kernels. Partitioning these critical kernels to hardware has been shown to provide application speedups of 10-100X [14][17][22][25]. Such approaches are effective because many software applications follow the 90-10 rule of thumb that states 90% of an application's execution time is spent executing 10% of the application's code.

While static profiling is feasible for many applications, dynamic profiling is essential for most dynamic optimization techniques. For example, warp processing dynamically and autonomously re-implements critical software kernels as hardware coprocessors within an on-chip FPGA [22]. As with many dynamic optimization approaches, warp processing relies on accurate dynamic application profiling to determine which software kernels are potential candidates for hardware implementation.

Most existing profiling approaches introduce non-negligible runtime overheads. For embedded systems, especially those with real-time requirements, this runtime overhead can adversely affect an application's execution. In the case of real-time systems, which are usually designed with very tight timing constraints, the slightest run time overhead can lead to missed deadlines and potential system failure. Hence, there is a need for dynamic, non-intrusive profiling techniques that provide accurate and detailed profiling statistics suitable for embedded and real-time systems.

In this paper, we present an efficient, non-intrusive dynamic application profiler (*DAProf*) capable of profiling an executing application by monitoring the application's short backwards branches and providing detailed loop execution statistics. In Section 2, we provide an overview of previous profiling approaches, specifically highlighting the non-intrusive, frequent loop detection profiler presented in [12]. In Section 3, we present our dynamic application profiler that provides detailed information regarding loop execution behavior, including the breakdown of loop executions versus average iterations per execution – providing both additional profiling information and improved accuracy compared to the frequent loop detection profiler. In Section 4, we highlight the area requirements, performance, and profiling accuracy of the *DAProf* design.

2. PREVIOUS WORK

A common software-based approach involves "instrumenting" the application by adding code to count frequencies of the desired code regions [13][16]. Software instrumentation is straightforward and flexible, yet incurs significant runtime overhead, especially if the granularity of the code regions being profiled is fine. To reduce runtime overhead, other profiling approaches use statistical sampling techniques [1][8][29]. Such methods either interrupt the microprocessor at certain intervals or create an additional software task for profiling, and then read the program counter and other internal registers to statistically determine execution behavior. Again, for embedded systems, especially those with real-time requirements, the slightest run time overhead can have significant impact on the application execution.

Other profiling approaches rely on hardware integrated within the microprocessor to assist software developers in profiling an executing program [7][23][24][28]. Such hardware-assisted profiling approaches utilize event counters or branch execution statistics to identify application hotspots [7] or frequently executed execution paths [23]. Although these hardware-assisted profiling approaches may incur lower overheads compared to software-based profiling methods, the runtimes overheads cannot be ignored and incur similar ramifications.

Of notable interest, is the frequent loop detection profiler that non-intrusively monitors the instruction addresses seen on the memory bus and profiles loop iterations by monitoring short backwards branches [12]. A short backwards branch is a branch instruction whose target address has a short negative offset and is typically used to branch backwards at the end of a loop. Whenever a short backwards branch occurs, the frequent loop detection profiler updates a small cache – perhaps just 16 entries – that stores the frequencies of the short backwards branches. When any of the registers storing the branch frequencies become saturated, the profiler shifts all cache entries right by one bit, thereby maintaining a list of relative branch execution frequencies while ensuring all branch frequency do not eventually become saturated.

For hardware/software partitioning approaches utilizing profiling to guide the partitioning process, the frequent loop detection profiler can provide a relative ranking of loops to guide the order in which loops are considered for hardware implementation. However, without further simulation or analysis, such limited profiling information may lead to suboptimal hardware/software partitioning results as performance improvements cannot be accurately estimated using only the relative execution frequency. The breakdown between loop executions and iterations per execution can have a significant impact on performance due to communication and synchronization requirements.

The speedup ($S_{HW/SW}$) after partitioning one or more loops to hardware can be estimated as follows:

$$S_{HW/SW} = \frac{T_{SW}}{T_{HW/SW}}$$
$$T_{HW/SW} = T_{SW} - T_{SW(loop)} + T_{HW(loop)} + T_{comm}$$
$$T_{comm} = Execs * \left(T_{Init} + T_{sync}\right)$$

where, T_{SW} is the software only execution time, $T_{HW/SW}$ is the execution of the partitioned application, $T_{SW(loop)}$ is the software only execution time of the loops being partitioned to hardware, $T_{HW(loop)}$ is the hardware execution time of the partitioned loops, and T_{comm} is the communication requirements for initializing and synchronizing with the hardware implementation. The communication time can be calculated as the number of times the hardware is executed (*Execs*) multiplied by the sum of the initialization time (T_{Init}) and synchronization time (T_{Sync}), where T_{Init} and T_{Sync} are the time required to transfer any required data between the software and hardware before and after the partitioned hardware loop execution. While the hardware execution time can be estimated using total loop iterations, the communication requirements depend on the number of times the loop is executed and can have a significant impact on overall speedup. Loops with larger executions and fewer iterations per execution will have greater communication requirements compared to similar loops with fewer executions but greater iterations per execution.

Consider an application in which two potential loops, *Loop A* and *Loop B,* have been identified as candidates for partitioning, but hardware resources are only available to implement one partitioned loop. As reported by the frequent loop detection profiler, *Loop A* has total iterations of 10,000 whereas *Loop B* has total iterations of 12,000. Furthermore, it is known or can be estimated that *Loop A* and *Loop B* account for 33% and 40% of total application execution time, respectively. Without additional information, a hardware/software partitioning approach will select *Loop B* to implement in hardware, as dictated by Amdahl's Law. However, if *Loop A* executes 5 times and iterates 2,000 times per execution, and *Loop B* executes 6,000 times and iterates 2 times per execution, the communication requirements of *Loop B* are 1200X greater and may severely impact the overall speedup. As a result, *Loop A* may be a better candidate for partitioning. As such, detailed loop execution information including loops executions and iterations per execution are essential in order to avoid suboptimal partitioning results.

3. DYNAMIC APPLICATION PROFILER (*DAPROF*)

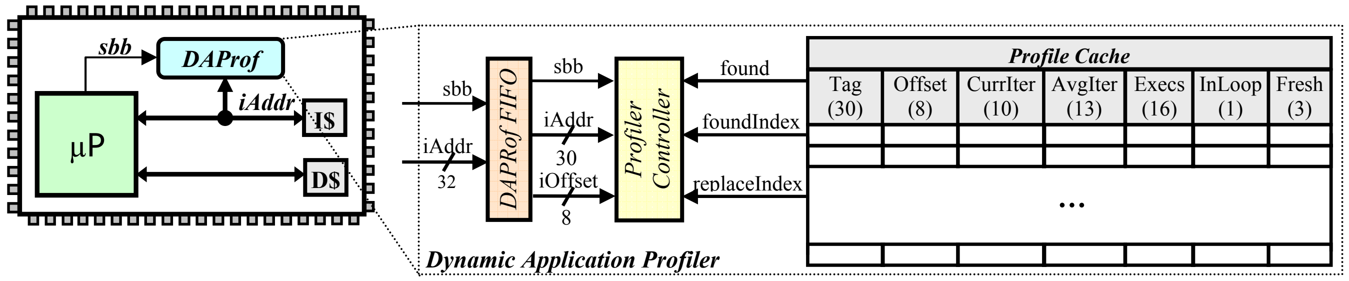
Figure 1 presents an overview of the dynamic application profiler (*DAProf*), highlighting its integration within a microprocessor based system and its internal profiling architecture. *DAProf* non-intrusively monitors the microprocessor's instruction bus to determine the address of the currently executed instruction whenever a short backwards branch is executed. The *DAProf* design considers a short backwards branch as a branch instruction whose target addresses is a negative offset of less than 1024, which corresponds to small loops containing less than 256 instructions. In comparison, the frequent loop detection profiler only considered loops with a smaller branch offset of 256, or 64 instructions. However, the most frequently executed loops within several of the applications considered in this paper are larger than would be supported by a branch offset of only 256, either as a result of complex loop functionality or as a result of loops that have been unrolled manually or during compilation.

While the *DAProf* could directly decode the instructions seen on the instruction bus, we currently assume the microprocessor provides a one bit output, *sbb*, indicating a short backwards branch has been executed. Such support would require minor modification to a microprocessors' decoding logic. *DAProf* design consists of a profiler FIFO for synchronizing between the microprocessor and profiler, a profile cache that stores all relevant profile statistics for those loops currently being profiled, and a profiler controller that analyzes the short backwards branches to update the profiling statistics within the profile cache.

3.1 Profiler FIFO

DAProf's profiler FIFO monitors the microprocessor's instruction bus and *sbb* output signal. Whenever a short backwards branch occurs, the profiler FIFO determines the branch instruction's address and offset and stores both values within a small internal FIFO. The *offset* is the number of instructions within the identified loop and is used along with the branch instruction's address to represent the beginning and end of each loop within the profiler.

Figure 1. Dynamic Application Profiler (*DAProf*) consisting of *Profiler FIFO*, *Profiler Controller*, and *Profile Cache*. *(Bit widths of profile cache entries shown within parenthesis)*.



In addition, the profiler FIFO is used to synchronize between the microprocessor and internal *DAProf* design because the microprocessor may operate at a higher clock frequency. Short backwards branches do not occur on every clock cycle. Thus, the internal *DAProf* profiler design does not need to operate at the same frequency of the microprocessor. Any meaningful loop will at least consist of two instructions in addition to the short backwards branch. Hence, short backwards branches are expected to occur no more than once every three instructions, implying the internal *DAProf* design can operate at one third the frequency of the microprocessor. However, the profiler FIFO should be large enough to accommodate bursts of short backwards branches that may occur periodically as the application executes. We experimentally determined that a FIFO with only four entries is sufficient for the applications considered in this paper.

3.2 Profile Cache

The profile cache is a small cache that maintains the current profiling results and intermediate information needed for loop identification, profiling statistics, loop execution monitoring, and determining which profile cache entry should be replaced when new loops are executed. We currently consider a 32 entry profile cache, which is sufficient for profiling the embedded applications considered within this paper.

3.2.1 Loop Identification

Profiled loops are identified within the profile cache by the address of the loop's short backwards branch, which serves as the *Tag* entry for the cache, and by the loop's *Offset* determined by the profiler FIFO. Considering a 32-bit ARM processor and byte addressable memory, the lower two bits for all instructions will be identical. Hence, the profile cache's *Tag* entry is a 30-bit entry that stores the most significant bits of a loop's short backwards branch address. The *Offset* entry is an 8-bit entry that corresponds to the number of instructions within the profiles loop. As described earlier, both the *Tag* and *Offset* are determined by the profiler FIFO and provide the mechanism for identifying the loop bounds.

3.2.2 Iteration/Execution Statistics

The main profiling information stored within the profile cache includes loop executions, average iterations per loop execution, and loop iterations for the current execution.

Loop *Executions* provide the number of times a loop has been executed throughout the application execution. As *DAProf* is intended to monitor an application over extended execution periods, regardless of the number of bits used to represent loop executions, the number of loop executions will eventually become saturated. *DAProf* utilizes a 16-bit entry for storing loop executions that allows 65,536 loop executions to be profiled without saturations.

The *Current Iterations* provides the number of times a loop has iterated for the current loop execution and is stored within the profile cache as a 10-bit entry. Thus, *DAProf* can accurately profile loops with a maximum of 1024 iterations per executions, which is well suited for most embedded applications.

The *Average Iterations* stores the average number of times a loop iterates per loop execution. As many loops do not iterate a fixed number of times per loop execution, the average iterations cannot be accurately stored as an integer value. Instead, the profile cache stores the average iterations as a 13-bit fixed point number using 10 bits to represent the integer portion and 3 bits for the fractional part. We note that the number of bits required to represent the integer portion of the fixed point number is equal to the number of bits used to store the current iterations.

3.2.3 Loop Execution Monitoring

The profile cache contains a 1-bit *InLoop* flag that is utilized to indicate the loop is currently being executed. The *InLoop* flag is essential in determining if the execution of a short backwards branch corresponds to a new loop execution or an additional iteration for the current execution.

3.2.4 Freshness & Replacement Policy

Although many replacement polices were considered and analyzed, including least recently used and estimated total instructions executed, the replacement policy incorporated within the profile cache uses total loop iterations to determine which entry within the profile cache will be replaced when a new loop is executed, where the entry with the lowest total iterations will be replaced. The total loop iterations are calculated as the product of the average iterations and executions. While this policy performs relatively well on its own, newly executed loops may not execute or iterate quickly enough to avoid being immediately replaced. We note that the frequent loop detection profiler also utilizes least total iterations as the replacement policy and exhibits this detrimental behavior when profiling several of the applications considered within this paper, as further detailed in Section 4.3.

To solve this problem, *DAProf's* profile cache includes a unique 3-bit loop *Freshness* value that represents how recently a loop has been executed or iterated, where a larger freshness indicates the loop has been more recently executed. The freshness value is utilized within the replacement policy to only consider loops for replacement if the loops are not considered fresh – a loop that is not fresh has a freshness value of zero. A 3-bit freshness entry allows seven loops to be considered fresh and allows newly executed loops to be profiled for an extended duration before their profile cache entry will be considered for replacement.

3.2.5 Associativity

The *Associativity* of the profile cache potentially provides tradeoff between cache size and cache performance with the accuracy of the profiling results. With a fully associative profile cache, the replacement policy must compare all entries within the cache to determine the entry with the smallest total iterations, thereby requiring large hardware resources and reducing the overall performance of *DAProf*. Decreasing the associativity of the profile cache provides increased performance and smaller area requirements by reducing the number of entries the replacement policy must consider. Finally, one must consider the relation between the profile cache's associativity and freshness. For a profile cache with a small associativity and large maximum freshness, all entries within the same cache set may be considered fresh and should not be selected for replacement. To avoid this potential problem, the maximum freshness should be no greater than one half of the profile cache associativity.

3.3 Profiler Controller

Figure 2 provides pseudocode of the dynamic profiling of *DAProf's* profiler controller. The profiler controller interfaces with the profiler FIFO and updates the profiling results for the current loops within the profile cache. The profiler controller receives the short backwards branch address (*iAddr*) and offset (*iOffset*) from the profiler FIFO in addition to a *found*, *foundIndex*, and *replaceIndex* signals from the profile cache. The *found* and *foundIndex* signals indicate if the current short backwards branch is found within the profile cache and at what location. The *replaceIndex* provides the index for the loop entry that will be replaced if the current short backwards branch is not found within profile cache.

Whenever a short backwards branch is available from the profiler FIFO, the profiler controller will determine if the loop is found within the cache. If the loop is found and the loop is currently executing – as indicated by the loop's *InLoop* flag – the short backwards branch execution indicates a loop iteration has been detected and the loop's current iterations are incremented. Otherwise, if the loop is not currently being executed, a new loop execution has been detected. For new loop executions, the profile controller increments the loop's executions, sets the *InLoop* flag, sets the current iterations to one, decrements the freshness for all other loops, and sets the freshness of the current loop to the maximum freshness.

Finally, if the profiler controller detects that the loop's executions is saturated, the executions for all loops will be divided by two – thereby handling executions saturations in the same manner as the frequent loop detection profiler handles total iteration saturations. However, in contrast to the frequent loop detection profiler, saturations within *DAProf* will only affect the accuracy of the loop executions without affecting the accuracy for average loop iterations per execution. In addition to ensuring the executions for any loop never becomes saturated, this approach provides a mechanism for monitoring the dynamic nature of an application in which older loops that were previously considered important may no longer be executed. For example, initially, a previously executed loop's high total iterations may ensure the loop is not replaced during profiling. However, after several saturations are encountered, the reported total iterations will be decreased relative to other loops and can be replaced if the loop is no longer executed.

If a loop's short backwards branch is not found within the profile cache, the profiler controller will replace the cache entry indicated by *replaceIndex*. The profiler controller initializes the

Figure 2: Pseudocode for *DAProf's* profiler controller.

```
DAProf (iAddr, iOffset, found, foundIndex, replaceIndex):
if ( found )
  if ( InLoop[foundIndex] )
    CurrIter[foundIndex] = CurrIter[foundIndex] + 1
  else {
    for all i, Fresh[i]         = Fresh[i] – 1
    Execs[foundIndex]           = Execs[foundIndex] + 1
    CurrIter[foundIndex]        = 1
    InLoop[foundIndex]          = 1
    Fresh[foundIndex]           = MaxFresh
    if  ( Execs[foundIndex] = MaxExecs )
      for all i, Execs[i] = Execs[i] >> 1
  }
else
  for all i, Fresh[i]           = Fresh[i] – 1
  Tag[replaceIndex]             = iAddr
  Offset[replaceIndex]          = iOffset
  CurrIter[replaceIndex]        = 1
  AvgIter[replaceIndex]         = 0
  Execs[replaceIndex]           = 1
  InLoop[replaceIndex]          = 1
  Fresh[replaceIndex]           = MaxFresh
}
for all i, if ( inLoop[i] && !(iAddr <= Tag[i] &&
                              iAddr >= Tag[i]-Offset[i]) ) {
  InLoop[i] = 0
  AvgIter[i] = (AvgIter[i]*7 + CurrIter[i])/8
}
```

profile cache entry by setting the *Tag* and *Offset* to those of the newly profiled loop's, setting the executions to one, setting the *InLoop* flag, setting the current iterations to one, decrementing the freshness for all other loops, and setting the freshness of the newly executed loop to the maximum freshness.

For all short backwards branches, the profiler controller checks all entries of the profile cache whose *InLoop* flag is set to determine if the application is still executing within those loops. If a loop is no longer being executed, the profile controller resets the *InLoop* flag and updates the loop's average iterations.

The average iteration calculation has a significant impact on the profiler's accuracy, hardware requirements, and performance. For example, a straightforward method for calculating average iterations computes the exact average using the equation:

$$AvgIter_i = \frac{AvgIter_i * (Exec_i - 1) + CurrIter_i}{Exec_i},$$

in which the average iterations are calculated as the previous total iterations – determined by multiplying the previous average iterations by the previous executions – plus the current iterations divided by the current executions. This method for calculating average iterations provides excellent accuracy across the entire application execution but requires floating point addition, multiplication, and division. Such an implementation would be too costly in terms of area and performance.

Alternatively, the profiler controller could perform the same calculation using integer multiplication and division. However, this approach leads to inaccurate profiling. As the number of executions increases, the denominator of the calculation will increase to the point that regardless of a loop's current iterations, the resulting average iterations will remain unchanged. Figure 3 presents the average iterations calculation for the floating point exact average iteration calculation, integer exact average iteration calculation, and the real average iterations over the past 100 loop executions for a frequently executed loop within the *cjpeg* application of the MiBench benchmark suite [15]. As demonstrated from the real average iterations over the past 100 loop executions, the loop being profiled has significant variation in iterations per execution. While the floating point exact average calculation provides the correct average iterations across the entire application execution, it is unable to provide any means for detecting or adjusting to such dynamic changes in execution behavior. On the other hand, the integer exact average iterations calculation is completely inaccurate. After several loop executions, the calculated average iterations remain constant for the remainder of the application execution.

Instead of relying on an exact average iteration calculation, the *DAProf's* profiler controller utilizes a weighted average in which the previous average iterations accounts for 7/8th and the current iterations account for 1/8th of the calculated average iterations, as provided by the following equation:

$$AvgIter_i = \frac{7 * AvgIter_i}{8} + CurrIter_i ,$$

in which the average iteration is calculated using a fixed point representation described earlier. This ratio based average iteration calculation can be efficiently implemented in hardware while providing excellent accuracy. Using the 13-bit fixed point representation to store the average iterations, this calculation is equivalent to:

$$AvgIter_i = \frac{7 * AvgIter_i + CurrIter_i}{8}$$

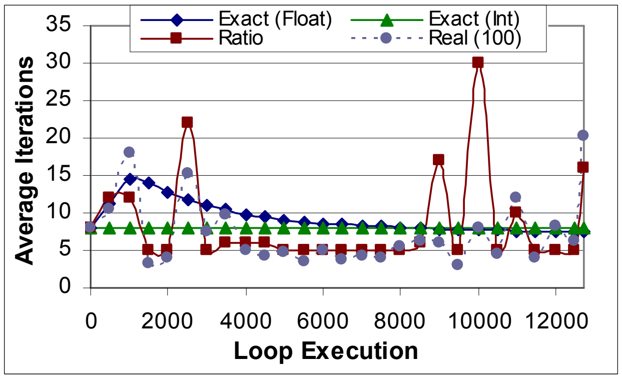
Figure 3 further presents the 7/8th ratio average iteration calculation for the selected loop within the *cjpeg* application. By providing a weighted average, the ratio average iteration calculation is able to capture dynamic changes in loop executions – most closely tracking the real average iterations. Although a 7/8th ratio is utilized within the current *DAProf* design, other ratios – such as 15/16th or 31/32nd – may be utilized to control how quickly or slowly the profiler will adapt to changing loop execution behavior with anticipated tradeoffs of area, accuracy, and speed at which the profiler design can adapt to such changes.

4. EXPERIMENTAL RESULTS

4.1 Area and Performance

We consider three alternative profiler implementations including a fully associative, 16-way associative, and 8-way associative *DAProf* designs. *DAProf* was implemented in Verilog and synthesized using Synopsys Design Compiler targeting a UMC 0.18 µm technology. For all implementations, the profiler FIFO has a maximum operating frequency of 934 MHz. Because the profiler FIFO can only execute three times faster than the profile cache and controller, *DAProf's* overall operating frequency is limited by profile cache and profiling controller. For a fully associative implementation, *DAProf* requires 107,477 gates (1.75 mm^2) and can execute at a maximum operating frequency of 415 MHz. The area required for the fully associative *DAProf* design is approximately 20% of the area of an ARM9 processor implemented within the same UMC 0.18 µm technology. The 16-way associative *DAProf* design requires 74,744 gates (1.22 mm^2) with a maximum operating frequency of 438 MHz. Finally, the 8-way associative *DAProf* design requires only 59,036 gates (0.96 mm^2) with a maximum operating frequency of 495 MHz. The 8-way associative *DAProf* design requires only 11% of the area of an ARM9 processor and is 45% smaller and 20% faster than the fully associative *DAProf* design.

Figure 3. Comparison of floating point exact average calculation, integer exact average calculation, 7/8th ratio average calculation, and real average over the past one hundred loop executions for a frequently executed loop within *cjpeg*.



4.2 Profiling Accuracy

To analyze the accuracy of the *DAProf* design, we compare the profiling results of *DAProf* with that of an accurate simulation based profiling method capable of fully profiling nested loop executions and iterations, function calls and executions, as well as recursive function calls [26]. We profiled the various consumer electronics applications provided within the MiBench benchmark suite [15] using the fully associative, 16-way associative, and 8-way associative *DAProf* designs.

For the top ten loops of each application, we analyzed the profiling accuracy in terms of percent error in reported average iterations, executions, and percentage of total application execution time, as presented in Figure 4 for a fully associative, 16-way associative, and 8-way associative *DAProf* designs.

The percent error in average iterations is calculated as the sum of differences between the reported and actual average iterations divided by the sum of the actual average iterations as follows:

$$AvgIter_{\%error} = \frac{\sum_{i=1}^{10} \left| AvgIter_{i(DAProf)} - AvgIter_{i(actual)} \right|}{\sum_{i=1}^{10} AvgIter_{i(actual)}}$$

On average, *DAProf* provided good profiling results with an error in reported average iterations of 11%, 11%, and 10% for a fully associative, 16-way associative, and 8-way associative implementations, respectively. In the best case, a 16-way associative *DAProf* design has an error of only 0.5% in reported average iterations for the application *tiffdither*. However, across all applications, the 16-way associative *DAProf* design has the lowest accuracy. For the applications *mad* and *tiff2rgba*, the reported average iterations exhibit an error of 35% and 30%. As further discussed in Section 4.4, this profiling inaccuracy is caused by function call interference that inadvertently results in the *InLoop* flag being incorrectly reset.

Because of unavoidable execution saturations, the loop executions reported by *DAProf* may not directly correspond to the actual total number of loop executions. Thus, the percent error in reported loop executions is calculated as follows:

$$Exec_{\%error} = \frac{\sum_{i=1}^{10} \left| \frac{Exec_{i(DAProf)}}{\sum_{j=1}^{10} Exec_{j(DAProf)}} - \frac{Exec_{i(actual)}}{\sum_{j=1}^{10} Exec_{j(actual)}} \right|}{10},$$

in which the number of execution for each loop is calculated as that ratio of the reported loop executions of each loop to the total loop executions of the top ten loops. On average, the *DAProf* design has an error in reported loop executions of only 3% for all implementations.

Finally, as the percentage of total application execution time is often utilized to determine the critical kernels of an application, we estimated the percentage of total application execution time for each profiled loop using the *DAProf* profiling results. The percent error in percentage of application execution time is simply the average absolute difference between the estimated and actual percentage application execution time for all top ten loops, calculated using the following equation:

$$\%ExecTime_{\%error} = \frac{\sum_{i=1}^{10} \left| \%ExecTime_{i(DAProf)} - \%ExecTime_{i(actual)} \right|}{10}$$

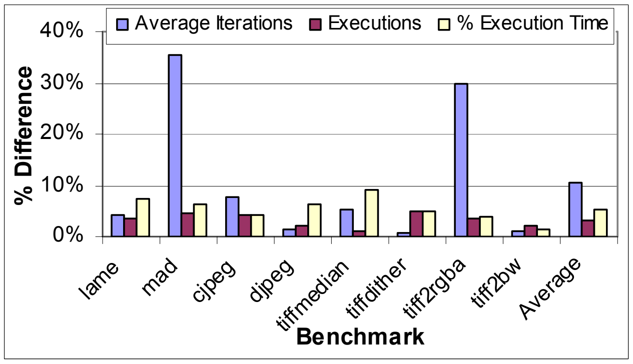
While function call interference may lead to errors in reported average iterations, the combined accuracy of average iterations and executions results in only a 5% error in the estimated percentage of application execution time for all *DAProf* implementations.

Overall, all three *DAProf* implementations performed equally well. While the fully associative design may be beneficial for some applications, for the applications considered, an 8-way associative *DAProf* design achieves similar accuracy with smaller hardware requirements and higher performance.

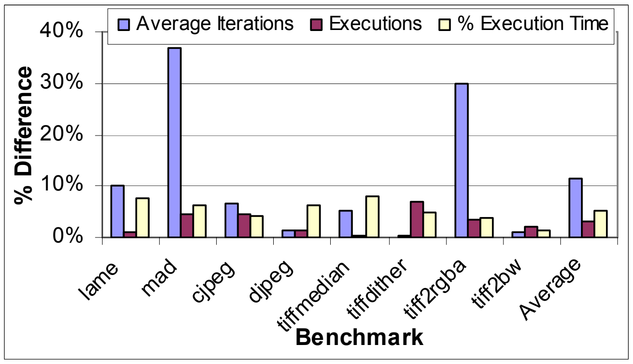
4.3 Comparison with Frequent Loop Detection Profiler

While *DAProf* provides additional profiling information beyond that available with the frequent loop detection profiler, the results from both profilers can be utilized to estimate each loop's percentage of total application execution. Figure 5 (a) presents the percent error in estimated percentage of total application execution time of an 8-way associative *DAProf* design compared to the frequent loop detection profiler for the various embedded applications considered. Although the frequent loop detection profiler presented in [12] utilized a short backward branch distance of 256, we consider a short backwards branch distance of 1024 in these results to provide a fair comparison between the two approaches. We further note that utilizing a short backward distance of only 256 would lead to reduced accuracy for the benchmarks considered. On average, *DAProf* provides a marginal increase in profiling accuracy of 95% compared to the frequent

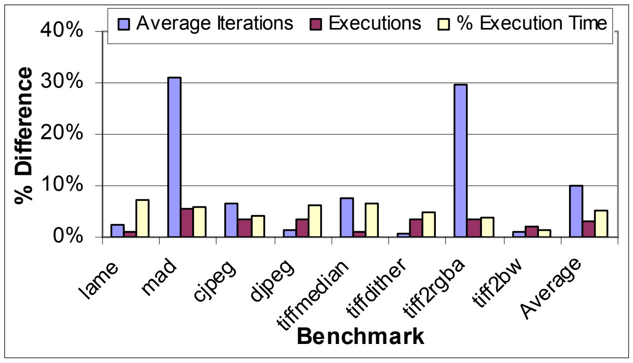
Figure 4. Percent error in average iterations, executions, and percentage of total application execution time of *DAProf* for MiBench consumer electronics applications.



(a) Fully Associative



(b) 16-way Associative

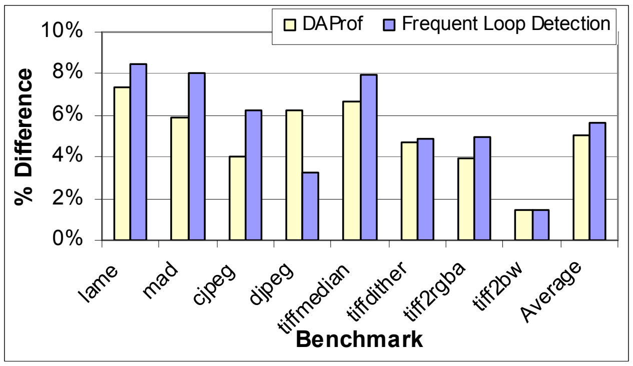


(c) 8-way Associative

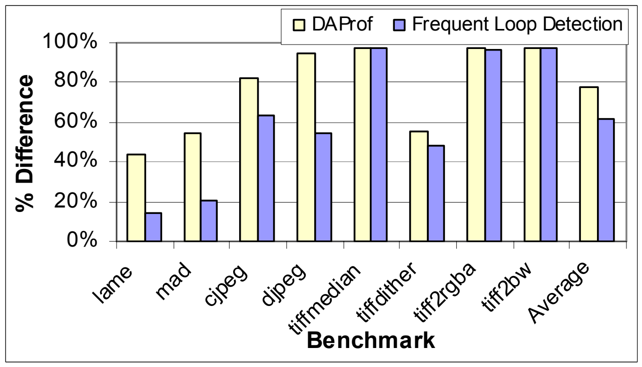
loop detection profiler's accuracy of 94%, with *DAProf* providing better profiling accuracy for all but one application.

More interestingly, Figure 5 (b) presents the percentage of total application execution time captured by the top ten profiled loops for an 8-way associative *DAProf* design compared to the frequent loop detection profiler for the various embedded applications considered. On average, *DAProf* is able to capture 78% of the total application execution time, compared to 62% captured by the frequent loop detection profiler. This increase is largely the result of *DAProf's* freshness calculation that ensures recently executed loops are not immediately replaced. For example, for the application *djpeg*, the frequent loop detection profiler does not capture the second most frequently executed

Figure 5. (a) Percent error in percentage of total application execution time and (b) percentage of total execution time captured for an 8-way associative *DAProf* versus the frequent loop detection profiler for MiBench consumer electronics applications.



(a) % Error in % of Total Execution Time



(b) % of Total Execution Time Captured

loop, which accounts for 23% of the total application execution time. As a result, whereas *DAProf* captures 94% of the total execution time for this application, the frequent loop detection profiler only captures 55% of the total execution time.

4.4 Function Call Interference

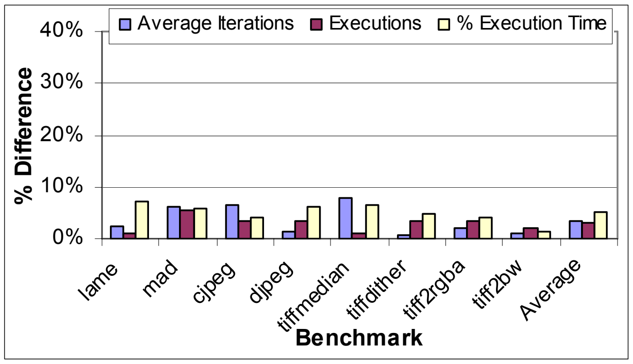
As previously mentioned, function call execution from within a loop currently being profiled can lead to profiling interference in which the *InLoop* flag of a currently executing loop is incorrectly reset. As illustrated in Figure 1, consider a loop, *Loop A*, that calls a function, *Func B*, where *Func B* also contains a loop, *Func B.1*. During execution of *Loop A*, the *InLoop* flag will initially be set and will remain set until a short backwards branch is executed outside of *Loop A*'s loop bounds. When the loop *Func B.1* is executed, the initial *DAProf* design will interpret the execution of *Func B.1*'s short backwards branch as indicating *Loop A* is no longer being executed. This execution behavior results in *Loop A*'s *InLoop* flag being reset and average iterations being updated. Furthermore, when *Func B* returns and *Loop A* iterates again, this iteration will incorrectly be interpreted as a new loop execution. Within the application *mad*, a loop that is affected by function call interference is incorrectly reported as executing 154 times with average iterations per execution of 252, whereas the correct loop executions and average iterations are 2772 and 14, respectively. Notice that the total loop iterations – calculated as the product of executions and average iterations – are identical.

Figure 6. Example of *Function Call Interference* in which the execution a loop's short backward branch within a function will lead to incorrectly resetting the *InLoop* flag of any loops in which that function is called.

```
Main() {                 FuncB() {
   ...                      ...

   Loop A {                 Loop FuncB.1 {
      ...                      ...
      FuncB ();             } // short backwards branch outside of
      ...                     // outside of Loop A's bounds
   }
                            ...
   ...                      return;
}                        }
```

Figure 7. Percent error in average iterations, executions, and percentage of total application execution time *(excluding loops with function interface)* for an 8-way associative DAProf for the top ten loops of the MiBench consumer electronics applications.



While we are currently developing an extended *DAProf* design with direct support for function call execution with very promising initial results, Figure 7 presents the percent error in average iterations, executions, and percentage of total application execution time for an 8-way associative *DAProf* excluding those loops affected by function call interference. For those loops not affected by function call interference, the 8-way *DAProf* design provides excellent accuracy, with errors of only 4%, 3%, and 5% for average iterations, executions, and percentage of total application execution time, respectively.

5. CONCLUSION

The dynamic application profiler (*DAProf*) provides an efficient, non-intrusive profiler capable of accurately profiling an application's execution. An 8-way associative *DAProf* can profile the application execution of a 495 MHz processor requiring only 11% additional area with a profiling accuracy of 90%, 97% and 95% for average iterations, loop executions, and estimated percentage of total application execution time.

Current and future work includes incorporating support for monitoring function calls and function returns to overcome the function call interference briefly mentioned within this paper in order to improve the overall accuracy of the profiling results with minimal additional hardware requirements. Initial results indicate that by monitoring both loop and function execution behavior, the extended *DAProf* design can achieve a profiling accuracy of 98%, 97%, and 95% for all profiling metrics. In addition, as many applications are multitasked and/or multithreaded, context switch interference will likely have similar detrimental affects of profiling accuracy. As such, future research efforts also include investigating efficient, minimally-intrusive, task-aware profiling methods for multitasked and multithreaded applications.

6. REFERENCES

[1] Anderson, J., et al. Continuous Profiling: Where Have All the Cycles Gone? 16th ACM Symp. of Operating Systems Design, 1997.

[2] Bala, V., E. Duesterwald, S. Banerjia. Dynamo: A Ttransparent Runtime Optimization System, Conf. on Programming Language Design and Implementation, 2000.

[3] Bellas, N., et al. Energy and Performance Improvements in Microprocessor Design Using a Loop Cache. ICCD, 1999.

[4] Calder, B., P. Feller, A. Eustace. Value Profiling. MICRO, 1997.

[5] Chung, E.Y., L. Benini and G. De Micheli. Automatic Source Code Specialization for Energy Reduction. ISLPED, 2001.

[6] Chernoff, A. Herdeg, M. Hookway, R. Reeve, C. Rubin, N. Tye, T. Bharadwaj Yadavalli, S. Yates, J. FX!32 A Profile-Directed Binary Translator, IEEE Micro, Vol 18, No. 2, pp. 56-64, 1998.

[7] Conte, T. M., Patel, B. A., Menezes, K. N., and Cox, J. S. 1996. Hardware-based profiling: an effective technique for profile-driven optimization. International Journal of Parallel Programming, Vol. 24, No. 2, pp. 187-206, 1996.

[8] Dean, J., et al. ProfileMe: Hardware Support for Instruction-Level Profiling on Out-of-Order Processors. MICRO, 1997.

[9] Diaconescu, A., A. Mos, J. Murphy. Automatic Performance Management in Component Based Software Systems, International Conference on Autonomic Computing, May 2004.

[10] Ebcioglu, K., E. Altman, M. Gschwind, S. Sathaye. Dynamic Binary Translation and Optimization. Transactions on Computers, Vol 50, June 2001.

[11] Gordon-Ross, A., S. Cotterell, F. Vahid. Exploiting Fixed Programs in Embedded Systems: A Loop Cache Example. IEEE Computer Architecture Letters, January 2002.

[12] Gordon-Ross, A., F. Vahid. Frequent Loop Detection using efficient Non-Intrusive On-Chip Hardware. IEEE Transaction on Computers, Vol 54, October 2005.

[13] Graham, S.L., P.B. Kessler, M.K. McKusick. gprof: a Call Graph Execution Profiler. Symposium on Compiler Construction, 1982.

[14] Guo, Z., Buyukkurt, B., Najjar, W., Vissers, K. Optimized Generation of Data-Path from C Codes. Design Automation and Test in Europe Conference (DATE), pp. 112-117, 2005.

[15] Guthaus, M., J. Ringenberg, D. Ernst, T. Austin, T. Mudge, R. Brown. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. Workshop on Workload Characterization, 2001.

[16] Hazelwood, K., A. Klauser. A Dynamic Binary Instrumentation Engine for the ARM Architecture. Conf. on Compiler, Architecture and Synthesis for Embedded Systems (CASES), 2006.

[17] Keane, J., C. Bradley, C. Ebeling. A Compiled Accelerator for Biological Cell Signaling Simulations. International Symposium on Field-Programmable Gate Arrays (FPGA), pp. 233-241, 2004.

[18] Klaiber, A. The Technology Behind Crusoe Processors. Transmeta Corporation, http://www.transmeta.com, 2000.

[19] Lakshminarayana, G., et al. Common-Case Computation: A High-Level Technique for Power and Performance Optimization. Design Automation Conference (DAC), 1999.

[20] Larus, R. James, Schmarr Eric. EEL: Machine Independent Executable Editing. Conference on Programming Language Design and Implementation, 1995.

[21] Lee, L.H., Moyer, B., Arends, J. Instruction Fetch Energy Reduction Using Loop Caches for Embedded Applications with Small Tight Loops. Intl. Symp. on Low Power Electronics and Design, 1999.

[22] Lysecky, R., G. Stitt, F. Vahid. Warp Processors. ACM Transactions on Design Automation of Electronic Systems (TODAES), Vol. 11, No. 3, pp. 659 - 681, 2006.

[23] Merten, M. C., Trick, A. R., George, C. N., Gyllenhaal, J. C., and Hwu, W. W. 1999. A hardware-driven profiling scheme for identifying program hot spots to support runtime optimization. SIGARCH Computer Architecture News, Vol. 27, No. pp. 136-147, 1999.

[24] Sprunt, B. Pentium 4 Performance Monitoring Features. IEEE Micro, Vol. 22, July 2002.

[25] Venkataramani, G., W. Najjar, F. Kurdahi, N. Bagherzadeh, W. Bohm. A Compiler Framework for Mapping Applications to a Coarse-grained Reconfigurable Computer Architecture. Conf. on Compiler, Architecture and Synthesis for Embedded Systems, 2001.

[26] Villarreal, J., R. Lysecky, S. Cotterell, F. Vahid. Loop Analysis of Embedded Applications. UCR Techn. Report UCR-CSE-01-03, 2001.

[27] Yellin, D. M. Competitive Algorithms for the Dynamic Selection of Component Implementations, IBM Systems Journal, Autonomic Computing, Vol. 42, 2003.

[28] Zagha, M., B. Larson, S. Turner, M. Itzkowitz. Performance Analysis Using the MIPS R10000 Performance Counters. Supercomputing, Nov. 1996.

[29] Zhang, X., et al. System Support for Automatic Profiling and Optimization. Intl. Symp. on Operating Systems Principles, 1997.