

Formulating and Implementing Profiling over Adaptive Ranges

Shashidhar Mysore Banit Agrawal Rodolfo Neuber Timothy Sherwood
Nisheeth Shrivastava Subhash Suri

Department of Computer Science
University of California, Santa Barbara
{shashimc, banit, rodolfo, sherwood, nisheeth, suri}@cs.ucsb.edu

Modern computer systems are called on to deal with billions of events every second, whether they are instructions executed, memory locations accessed, or packets forwarded. This presents a serious challenge to those who seek to quantify, analyze, or optimize such systems, because important trends and behaviors may easily be lost in a sea of data. We present Range Adaptive Profiling (RAP) as a new and general purpose profiling method capable of *hierarchically* classifying streams of data efficiently in hardware. Through the use of RAP, events in an input stream are dynamically classified into increasingly precise categories based on the frequency with which they occur. The more important a class, or range of events, the more precisely it is quantified.

Despite the dynamic nature of our technique, we build upon tight theoretic bounds covering both worst-case error as well as the required memory. In the limit, it is known that error and the memory bounds can be *independent* of the stream size, and grow only linearly with the level of precision desired. Significantly, we expose the critical constants in these algorithms and through careful engineering, algorithm re-design, and use of heuristics, we show how a high performance profile system can be implemented for Range Adaptive Profiling. RAP can be used on various profiles such as PCs, load values, and memory addresses, and has a broad range of uses, from hot-region profiling to quantifying cache miss value locality. We propose two methods of implementation of RAP, one in software and the other with specialized hardware, for which we also describe our prototype FPGA implementation. We show that with just 8k bytes of memory, range profiles can be gathered with an average accuracy of 98%.

Categories and Subject Descriptors: C.3 [Computer Systems Organization]: Special-Purpose and Application-Based Systems

General Terms: Algorithms, Design, Measurement, Performance

Additional Key Words and Phrases: Profiling Hardware, Range Adaptive, Value Locality

Author's Address: Shashidhar Mysore, Department of Computer Science, University of California, Santa Barbara, CA, 93106. Email: shashimc@cs.ucsb.edu

Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.

© 2007 ACM 0000-0000/2007/0000-0001 \$5.00

1. INTRODUCTION

Many proposed run-time systems rely on profile information to make informed design and optimization decisions. Procedure and data placement, trace scheduling, value specialization, network load balancing, dynamic compilation, and a whole host of power management techniques can all be guided by an accurate picture of what a program is doing and how it is interacting with the system. A major problem in dealing with streams of profile data generated is that we can only store a small amount of information yet we need to be able to accurately characterize the behavior of the entire stream. This is especially problematic if the profile information is not completely dominated by a small number of frequently seen, or “hot”, events.

A significant difficulty in gathering run-time profiles is keeping track of this data in a manner that requires little storage, incurs limited or negligible slowdown, and provides a consistent, accurate, and useful summary of the data. Profiling a large program for an extended amount of time (minutes or even hours), as required in a real system, results in the generation of huge amounts of data. Dealing with these large profiles in software requires clever schemes for adaptively sampling [Hirzel and Chilimbi 2001], compressing [Zhang and Gupta 2004] and compacting [Larus 1999] profiles to reduce the impact on memory.

The aim of our research is to explore a new profiling method capable of summarizing profile data in a streaming fashion (one-pass) with only a small and bounded amount of memory. *Range Adaptive Profiling* (RAP) uses a small set of counters to track *ranges* of profile data such as blocks of data and IP-addresses, segments of code, or ranges of load values. Every piece of data fed into the system is accounted for in some range (RAP merges the data rather than sample or filter), but the ranges which are chosen for profiling are adjusted dynamically based on observed program behavior. While perhaps not every type of profile can be merged easily into adaptive ranges, hot code regions can be found to guide optimization, ranges of values can guide encoding decisions and value prediction, while ranges of data memory will correspond to instances of data structures. Other types of profiles, such as edge profiling, can also be mapped onto adaptive ranges with simple extensions to the method. In Section 5 we describe three example uses of RAP in more detail.

In particular our paper makes the following contributions:

- We present the idea of Range Adaptive Profiling and show how it can be used to generate online summaries of different types of profile data including code, load values, memory content, and narrow-width operands.
- We describe Range Adaptive Profiling Trees and show how optimizing the branching factor and merging behavior can provide an implementable solution with guarantees on both summarization error and bounded memory.
- We describe the multi-bit trie based software implementation and the APIs provided along with the open-source Range Adaptive Profiling software.
- We present a method by which Range Adaptive Profiling can be efficiently pipelined if specialized hardware support is added and provide a detailed analysis of the required hardware. We also elaborate on our prototype implementation of RAP on an FPGA.
- We quantify the Range Adaptive Profiling error and memory requirements for

hot code regions and load value ranges. With as little as 8k bytes of memory, accuracy of up to 98% is possible.

The rest of the paper is laid out as follows: In Section 2 we begin by describing our online algorithm, while implementation details of our design are discussed in Section 3. In Section 4 we describe the pipelined hardware implementation prototype of Range Adaptive Profiling on an FPGA. In Section 5 we quantify the advantages of our scheme and provide some qualitative evidence of its usefulness in the form of range profiles. We describe related prior works in Section 6 and finally conclude in Section 7.

2. PROFILING WITH ADAPTIVE PRECISION

The first difficulty in building a run-time profiling system is in gathering the raw data. Several software techniques, such as binary instrumentation [Buck and Hollingsworth 2000; Luk and et. al. 2005; Srivastava et al. 2001; Srivastava and Eustace 1994; Bus et al. 2004] and sampling [Arnold and Ryder 2001], can be used to generate and analyze this profile information with only a moderate amount of overhead [Arnold and Ryder 2001; Ball and Larus 1996; Calder et al. 1997; Chilimbi 2001; Chilimbi and Hirzel 2002; Duesterwald and Bala 2000; Hirzel and Chilimbi 2001; Larus 1999]. Recently, several researchers have proposed various forms of architectural support [Anderson et al. 1997; Conte et al. 1996; Conte et al. 1994; Dean et al. 1997; Heil and Smith 2000; Narayanasamy et al. 2003; Peri et al. 1999; Sastry et al. 2001; Yang and Gupta 2002; Zilles and Sohi 2001] with the aim of increasing accuracy and further reducing the overhead of software based techniques. Value profiles can be exploited to perform code specialization [Calder et al. 1997], value prediction [Lipasti and Shen 1996; Zhou et al. 2003], and value encoding [Yang and Gupta 2002; Yang et al. 2000]. Operand profiles identify the potential to apply power and performance optimizations [Loh 2002; Brooks and Martonosi 1999]. Address profiles have been used for data layout optimizations [Rubin et al. 2002] and data prefetching mechanisms [Chilimbi and Hirzel 2002], and code profiling for focusing optimization efforts on the most important regions of a program. Control flow traces and path profiles [Ball and Larus 1996; Larus 1999; Zhang and Gupta 2001] can be used to perform path sensitive optimizations [Gupta et al. 1998; Young and Smith 1998] and path sensitive predictions [Jacobson et al. 1997]. A general purpose framework for dealing with profile data has even been proposed [Zhang and Gupta 2004]. While gathering data is a difficult problem, it is not the end of the story.

To explain the concept behind range adaptive profiling let us start with a simple example. Suppose we would like to know something about the regions of code that *gcc* is spending its time in. The simplest and lowest precision way to quantify this is to keep one counter which counts all instructions executed on behalf of *gcc*. The counter keeps a perfectly accurate count, and covers the entire program, but of course the profile has *no precision* and fails to provide any information on which subset of instructions is really the most important. If two counters are available, the next logical step might be to have one count the “top” half of *gcc* code and to have the second counter track the “bottom” half of *gcc* code. In this example each counter is tracking a range of code in *gcc*, although as we discuss in Section 5,

this works equally well for memory addresses, values, and other range based profile types. This idea of dividing the code into N ranges for N counters could be easily extended to 4, 8, 16 counters and so on. Unfortunately, this quickly gets out of hand, and to track the program at the precision of an instruction we would need counters for each and every basic block.

Our technique is based upon the realization that not all profile information is equally valuable. The more frequently a set of events occurs, the more important it is to precisely quantify and characterize this set of events. Specifically, it may be sufficient to group profile data into *ranges* - where the most frequently occurring ranges of events are identified and broken into more precise ranges while the least frequently occurring events are kept as larger ranges. If the profiling ranges are properly managed over time, we can strike a balance between profile resolution and overhead.

When a particular range of events constitutes a significant portion of the total profile, then that range should be sub-divided and profiled more precisely. This recursive refinement of profile ranges maps nicely onto a tree, where the root of the tree represents the entire range of events and each child of a node represents a refinement of the profiling for a particular subrange. We formalize this idea as *Range Adaptive Profiling* and show how we implement this idea in a specialized hardware scheme.

2.1 Profile Trees

To gather profiles where the granularity is changing dynamically, we will need a data structure in which we can store our profiles. The majority of the past work in this area has assumed a flat storage of the profile. Whether the data was gathered through hardware performance counters [Anderson et al. 1997], stratified sampling [Sastry et al. 2001], or even potentially in fixed ranges [Zilles and Sohi 2001; Zhou et al. 2004], the end result is essentially a list of equivalent items and their counts. While there exists some specialized software and hardware systems that attempt to tightly compress particular types of traces [Anderson et al. 1997; Conte et al. 1996; Conte et al. 1994; Narayanasamy et al. 2003; Sastry et al. 2001; Zilles and Sohi 2001], we believe we are the first to present a general hardware-based methodology for storing profiles in a *hierarchical* fashion.

As we mentioned above, the most natural way to store our hierarchical profiles is with a tree. This tree will keep a constantly up-to-date summary of the data stream, and in this section we describe the three types of operations on the tree that we need to support. The first, and by far the most common, operation is a simple *update*, where a counter in the tree is simply incremented to track the incoming data. To refine the granularity of a sufficiently hot range and to ensure precision, we have a *split* operation. Finally, we need *merges* to combine together relatively unimportant data which ensures that the tree is carefully pruned to maintain the least number of counters necessary to capture all the important information. The splits and the merges change the structure of the tree and hence dynamically re-map profile events to the counters. While at a high level this simply sounds like a simple tree, in reality each of these three functions has been specially designed such that the overall data structure is both implementable in an online and pipelined way and provides a type of worst case bound on error (the ϵ -error discussed in

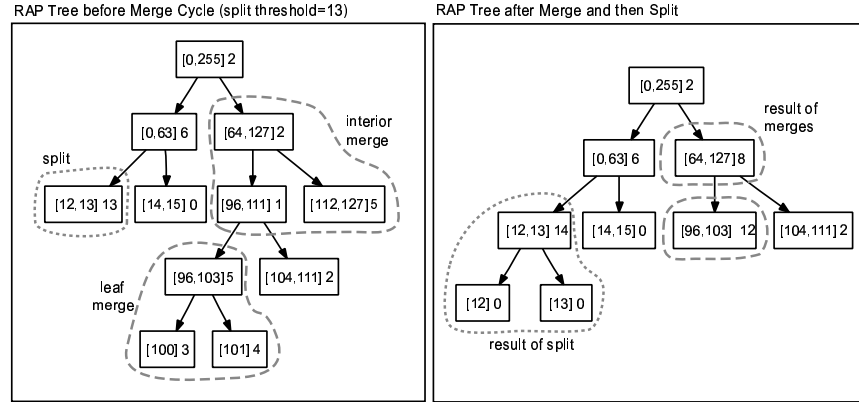


Fig. 1. A range adaptive profiling tree (in this example, each node has 2 out edges). The diagram on the left is the state of the tree just before a merge cycle begins. During a merge cycle, the tree is walked and any set of nodes that have insufficient weight to warrant separate profiles are *merged* (in this example the cutoff is a cumulative weight of 13). Following this merge, an access to item 12 might occur, which will push the node that captures the range [12,13] to go over the split threshold. This would cause the node to be *split* into two different nodes, and subsequent accesses to item 12 or 13 will be recorded on an item by item basis.

Section 2.2). While all three operations are discussed in detail, we begin with a discussion of the simple update.

The profile tree is built of nodes, and each node corresponds to a particular *range* of events that the profiler might see. As was mentioned earlier, the root node represents the entire range of events possible, and each child of a node will capture a proper sub-range of its parent. When an event enters the profiling system, for example the PC of a cache miss, this event is matched into the range that covers it. Because an event often matches several possible ranges, we need to find the *smallest* range that includes that event and then increment that node's counter. This will ensure that profiling is done with as much precision as possible without modifying the tree. As an example, please refer to Figure 1. This figure is a snapshot of the tree structure with each node in the tree tracking a range of values $[min_range, max_range]$ and a *count* to track the number of times an event entering the profiler mapped to this node as the smallest range covering the event. (Split and Merge operations referred to on this figure are explained in the following subsection). If an incoming event had a value of 12, in the graph on the left hand side, it would match the ranges of [0,255], [0,63], and [12,13] but only the node responsible for profiling [12,13] (smallest range) would have its counter updated.

2.2 Growing the Trees

While updates are the most common operation performed on a tree, updates do not modify the structure of the tree to adapt to the input stream. The two operations that actually modify the tree structure are *split*, which further refines the profiling

of a given range, and *merge*, which decreases the granularity with which a range is profiled. The main idea behind a split is that if a range is important enough, its counter will increase faster than average. Eventually this node will grow so ripe, that it makes sense to burst the node into a number of subranges. In this way the tree grows to increase precision where the profile has more weight. If the counters from a set of ranges are no longer a sufficient portion of the whole stream they can be merged together with little impact.¹

The key to applying the split and merge operations is knowing *when* they should be applied. This can be done by setting a *SplitThreshold*, and any node that grows larger than this threshold should sprout children to more accurately profile each of the sub-ranges. The *SplitThreshold* is a function of the number of events processed n , and the maximum possible height of the tree $\log_b(R)$ where R is the maximum range to be considered, and b is the branching factor of the tree. If a range is not fully refined yet, an event occurrence will be accounted for by one and only one range which is the smallest superset of the newly encountered event. This initial accumulation of the count along the path of the RAP tree towards the fully refined range is the culprit which gives rise to the error in counting. If we build a hierarchy tree over a given maximum range R , the height of this tree and hence the number of ancestors of any node is at most $\log_b(R)$. To minimize this error, we must limit the count of these ancestor nodes by setting appropriate split threshold. Specifically we set

$$SplitThreshold = \frac{\epsilon \cdot n}{\log_b(R)}$$

where ϵ is a user defined constant between 0 and 1. If the *SplitThreshold* is set in this way, the *maximum* amount of error possible, relative to the entire input stream², is ϵ . For example, if the user sets ϵ to 1%, that means for any given range the estimate for that range will never be off by more than 1% of the total events processed. Further, it can be shown that the maximum amount of memory required by a tree built with this split threshold is $O(\log_b(R)/\epsilon)$ (For a more formal and detailed treatment of the proofs please refer to [Hershberger et al. 2004]). The exact byte counts, overheads, and percent errors are described in Sections 3 and 5.

Split - Calculating when an update needs to be followed with a split operation is actually a fairly straightforward task. We simply compare the value of the counter with the split threshold described above. Any time a node grows over this limit, we need to add a set of children to this node that cover and equally subdivide its range. The original node keeps its counter, and each of the children have their counts initialized to zero.

Merge - While splitting is crucial to the adaptation of the granularity, if all we ever did was update or split, it would be impossible to bound the total amount of memory required. For example, a region of code may start out “hot” and as such might have been split into many separate counters to count every basic block within. Later, however, it may turn out to be relatively unimportant and we may

¹Counters are never decremented which is why this is not a sampling scheme, rather merges happen when the rest of the tree has outgrown a particular set of regions.

²The ϵ error is defined as a fraction of the total length of the input stream, while percent error is error relative to the actual count of a range

wish to release all counters associated with the basic blocks and retain just one counter for the entire region. A way to un-split a set of regions is to do a merge operation. Rather than simply throwing away the range profile information from each of the children nodes, we incorporate them into the parent node. Because any count gathered for a child is equally valid to be stored on the parent range (because it is a super-range), we simply sum together the count of the child nodes and add them into the parent.

3. IMPLEMENTATION DETAILS

In order to build an effective profiling system around the algorithm described in Section 2, there are several tasks that need to be performed at runtime. First off, we need a mechanism to gather profile data. In a purely software based approach, these profiles can be generated through either binary instrumentation [Buck and Hollingsworth 2000; Luk and et. al. 2005; Srivastava and Eustace 1994; Srivastava et al. 2001; Bus et al. 2004] or hardware performance counters [Anderson et al. 1997; Corporation 1995; 1997; Hewlett-Packard 1994; Inc 1995]. Even in a software based approach the input data should be buffered to some extent and duplicate values should be merged together to help improve performance. In the case of a hardware-assisted or hardware-only approach, we assume that the profiles are generated using a pre-existing or proposed profiling structure [Conte et al. 1996; Dean et al. 1997; Heil and Smith 2000; Zilles and Sohi 2001]. Specifically, we assume a structure similar to ProfileMe [Dean et al. 1997] for collecting the input events (load values, PC, memory addresses, etc.). The buffered events are processed one after another in the order the load instructions (for value profiling) or branch instructions (for code profile) retire. The buffer size and the sampling module will affect the overall accuracy of a profile, but it has no impact on the way in which Range Adaptive Profiling summarizes the data and for this paper we concentrate solely on the accuracy of the summarization step.

Processing the gathered events and maintaining the RAP tree based counter structure can be time consuming if implemented naively. In this section we discuss ways of speeding up these tasks and describe the software implementation of RAP and also a hardware based approach that operates with little or no support from software.

3.1 Algorithm and Architecture Design Issues

To enable efficient storage and searches on the profile tree, a suitable *branching factor* (b) must be used. The branching factor is the number of children that will be generated in a split operation. If b is too small, the ranges marked for profiling will take longer to converge on the best set. For example, if one particular value in a range is accounting for 100% of the profile data seen, it will take exactly $\log_b(R)$ splits to finally start profiling this item individually which in turn effects the error in the profile. On the other hand, if b is too large, the amount of memory required to store the tree will grow. The higher the branching factor, the more extraneous children will be kept around. To seek a balance between these two constraints, we analyzed the effect of branching factor on the worst case number of nodes that can appear in the tree. Figure 2 shows this tradeoff. On the x -axis we have a variety of different branching factors, and on the y -axis is the worst case number of nodes

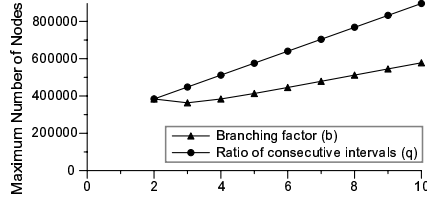


Fig. 2. The two independent graphs plotted show the memory size requirement for different branching factors b (lower graph) and merge-interval ratios q (upper graph). We choose $b = 4$ as it is a better trade-off between memory consumed and the height of the tree. With $q = 2$ we see that the memory size is the least. (With $R = 2^{32}$)

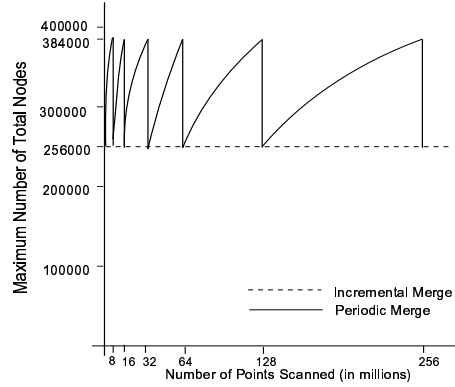


Fig. 3. If merges are performed continuously, the tightest bound on the number of nodes required can be maintained but at the cost of continuously searching for merge opportunities every single cycle. Instead, we can still have bounded worst case memory requirements if we batch the merges together with an exponentially decreasing frequency. (With $R = 2^{32}$)

that could be generated for a branching factor of b and an ϵ of 1%. We found that a branching factor of 4 provides a good tradeoff between the required amount of memory and the effect on performance and error. Note that the figure shows *worst case* number of nodes and as will be shown in Section 5, in the common case the number of nodes is a factor of 1000 less.

Another problem that shows up in an implementation is when to perform the merges. Finding a node that needs to be split is easy, a counter is updated and then we check if the counter is over the threshold. Finding the places where a merge must be performed is much more difficult, as they, by definition, happen away from where the updates are occurring. How does one detect when a merge is needed? One approach is to build a secondary merge heap, which stores a list of those nodes that are most in need of merging. While this approach is suitable from a theoretical standpoint and for a software implementation, updating the merge heap requires many extra tree operations and a full additional tree. Furthermore, one merge can result in a new node which in turn needs to be merged into its parent and so on. Rather than detecting and handling merges at the soonest possible time, we propose batching the merges together enabling an easier hardware implementation.

By performing merges periodically, instead of in a continuous manner, we avoid the problem of having to continuously search the tree for valid sets of nodes to be merged. In order to grow, the tree must split, and in order to split, the counts of the nodes must grow past the split threshold. The key point to see is that as the number of events processed grows, the relative rate at which the tree can split *must* slow down. In fact, an un-merged tree can grow at a rate which is at most

logarithmic with the number of events processed. Instead of having a fixed period for merges, we can have merges with an *exponentially increasing period* and the worst case bounds will still hold. This idea can be seen most clearly in Figure 3.

In Figure 3, the *x*-axis shows the number of events processed (instructions executed, values profiled, etc.). The *y*-axis is the worst case bound on the number of nodes required to profile with an ϵ of 1%. At the beginning, and after every merge, the worst case number of nodes is bounded to $384k$. After a merge, the worst case size of the tree grows slowly, inching up at a logarithmic rate. If we wait for some number of events e to pass before doing a merge operation, the next time around we can wait a total of $2e$ events before the worst case number of nodes grows to the same point. In other words, if it took e events to force a split in the first period, in the second period the tree will be twice as big and it will require twice as many events in order for a split to be necessary. While in this example, we double the interval between consecutive merges, in general, we could increase the interval by a factor of q . In figure 2 we show that doubling the intervals is the most cost effecting setting for q . Also, we need to note that the error with which RAP profiles is purely based on the criteria to split and has nothing to do with *when* we merge. Hence, the error guarantees with a periodic merge remains the same as the one where a merge happens on every event.

3.2 Software Range Profiling

More often than not, system profiling involves gathering different types of profile information, analyzing individual streams of profiles and correlating and/or deriving inferences about system behavior. These individual streams could be program counter values, memory load addresses, load values, register values, control flags, branch targets, or many others. Using the results from the previous sections, we have developed a software implementation of RAP that can be called from software-only systems. We now describe some of the APIs the implementation provides and how range adaptive profiler can be extended for simultaneous profiling.

The three main methods - `rap_init()`, `rap_add_points()`, `rap_finalize()` can be used for both online and offline profile trace analysis.

`rap_init(number_of_simultaneous_profiles)` - initializes the RAP tree with an initial set of counters and appropriate range values. `rap_init` also initializes data structures to enable profiling multiple events simultaneously. For example, to find both hot regions in a code and the most frequently accessed memory locations, RAP tree can be initialized to profile both program counters and load memory addresses. `number_of_simultaneous_profiles` specifies the number of distinct *types* of profiles that needs to be analyzed simultaneously.

`rap_add_points(profile_event_identifier, count, profile_type)` - Since RAP is a dynamically allocated tree, `rap_add_points` looks up the appropriate counter, updates the counter, and when needed calls the internal functions `rap_split()` and `rap_merge()` to either split or prune the tree to maintain adaptive precision with a limited set of counters. `profile_event_identifier` for a given `profile_type` is looked up in the RAP tree and its corresponding counter is incremented by `count`.

`rap_finalize()` - Some feedback directed optimization algorithms, such as those for value specialization, would like to know of any changes in program behavior. For this, they may want to query the profiler periodically or on an event-driven basis to

obtain the latest inference about the program. `rap_finalize` provides precisely this feature. Additionally, post processing phase of deriving statistical inferences about the stream can also be done through `rap_finalize`. The RAP tree which contains precise information can be dumped in an *ascii* format for further processing such as identifying hot-spots, range coverage, phase identification, and so on.

3.2.1 Optimizing RAP. Given that run-time system profilers need to process billions of events per second, it is very important to consider optimizations that increase the effective throughput of RAP. In this section, we propose the addition of a *merging event buffer* and examine the performance advantages quantitatively. For all experiments in this section, we choose to use a set of applications which would be a good mix representative of integer, floating point, and multimedia benchmarks - *gcc*, *gzip*, *vpr*, *applu*, *mpeg2 encode*, *mpeg2 decode* with Minnespec inputs [KleinOsowski and Lilja 2002] and ran them all to completion on a Pentium 4 (IA32) running Linux 2.4 (Fedora Core). We used Pin [Luk and et. al. 2005] as a binary instrumentation tool to gather the profile data. The RAP APIs were exercised by calls that were dynamically inserted into the host application, and the runtime of the complete system is examined. The merging event buffer has two significant advantages.

The first advantage of the merging buffer is that it reduces the behavior switch overhead. For a high precision run of RAP, the trees can grow to be quite large. The fact that every event in the host program is interleaved with an estimated set of about 20 accesses to a recursive data structure wreaks havoc on any locality in the original program. While this will not effect the functional measurement of the stream (because Pin does not instrument itself), it will certainly make the whole system execute much more slowly. Instead, by buffering the updates (which requires walking over only a small amount of memory) and committing them as a group, the impact on the memory hierarchy is greatly reduced. In fact, because they are committed as a group the total number of walks through the tree can be reduced as well, while the precision and behavior of RAP remains the same since we check for the need to split a node after every increment of a counter.

The second advantage is in exploiting locality of access in the profile events themselves. While RAP is built to allow efficient profiling regardless of whether there is any locality or not, the fact is that a few frequent event types are responsible for a strong majority of accesses. For example, in our observations of value profiles for *gcc*, the value zero accounted for 34% of the loads in the system. Clearly, calling RAP and traversing the tree to find the node for zero each and every time a load returning the value of zero is found would be wasteful. Instead, the merging buffer uses hashing to merge together *identical* events for performance reason before they are inserted into RAP. If there is a conflict in the hash table, instead of pointer chaining we evict the older entry from the hash table and then insert it into the RAP tree. For frequently occurring events, such as the zero value load in *gcc*, it will consistently be serviced by the fast software buffer and will only be committed to RAP tree when there is a conflict or at program completion. We have found that even small amounts of aggregation that happens due to the buffer can significantly improve the performance of the profiler, even when the profile stream was not expected to have a significant amount of locality (for example value profiling).

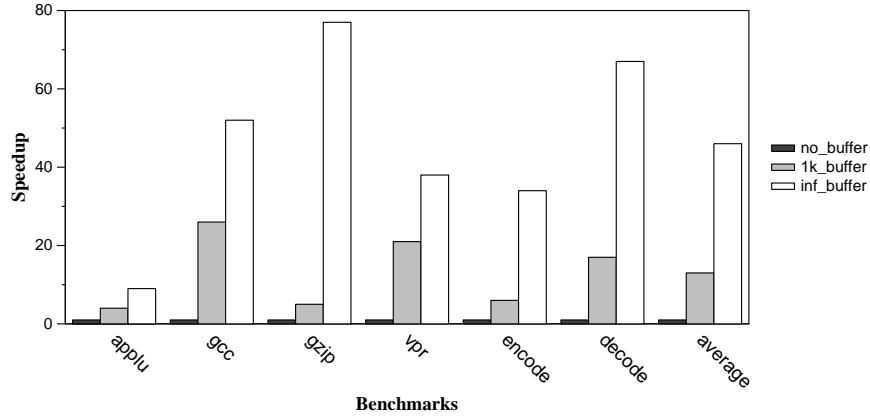


Fig. 4. The graph shows the speedup achievable due to a buffer to aggregate and collect events. The x -axis shows a set of benchmarks we choose to demonstrate the effects of a buffered RAP. The y -axis shows the speedup achievable in comparison with a non-buffered version of RAP. The third bar represents the possible speed up when all events are aggregated to the maximum extent (in cases where we have huge buffers or while processing offline traces).

Figure 4 shows the speed up achievable on different benchmarks shown on the x -axis. We can see that with just a 1k buffer, speedup of more than 25 times can be achieved on some benchmarks (*gcc*) with an average speedup of 13x. With a 1k buffer, RAP tree does not have to process every time a profile event enters the system, rather would process batches of identical events aggregated together. We also wanted to measure the best possible speedup that can be achieved by having a huge event buffer. To this end, we gathered the profile as the programs executed, and this trace was then compressed offline to the maximum extent by aggregating all identical events across the entire trace. RAP APIs were then called in to process these compressed traces. We observed that for *gzip*, a speed up of more than 70x was possible and an average speedup of 46x across all benchmarks.

3.2.2 Mutli-bit Trie Implementation. To efficiently implement a data structure that performs the job described in Section 2, we build on the idea of a multibit trie [Sanchez et al. 2001; Lampson et al. 1999]. A trie is an ordered tree data structure where the position of a node in the tree shows what key that node is associated with. All the descendants of any one node have a common prefix of the string associated with that node, and the root is associated with the empty string. A k -bit multibit trie is a trie defined over the alphabet $\{0, 1\}^k$, where each node then has 2^k children. If we are profiling over the set of 32-bit integers, then a k -bit trie will have a maximum height of $32/k$.

In order to use a multibit trie, we need to add some additional constraint to the problem definition from Section 2. Specifically, we need to require 1) that all of the ranges are of a size that is a power of two and 2) that all of the ranges are properly aligned (that is $range.start \bmod range.size = 0$). If these two properties hold then

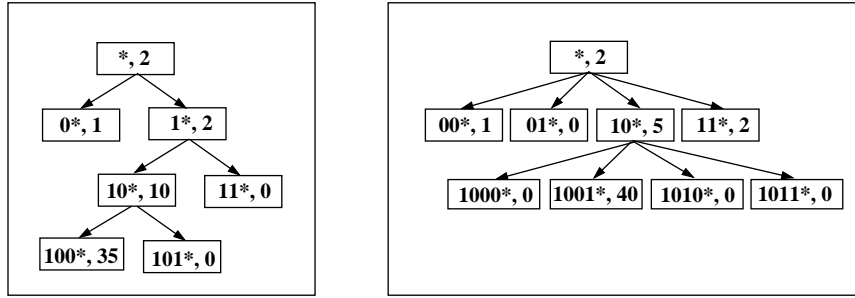


Fig. 5. The figure shows the trie-based implementation of the RAP tree. Every node of the tree is a counter with a prefix and a count. Any time an incoming profile event’s identifier hits the longest prefix match, the counter in the node corresponding to the match is incremented. The figure on the left is a binary trie where for every incoming profile event the counter is chosen by traversing down the RAP tree considering one bit at a time. The multi-bit trie version of the same is shown on the right hand side. We can see how RAP adapts to the most frequently seen events. (In this case it is $0x10010110$)

we can use a multibit trie to store our nested ranges, and we can exploit the existing work on efficient software multibit tries from the study of ip-lookup [Sanchez et al. 2001; Srinivasan and Varghese 1999a] to build a fast implementation.

As explained earlier in this section, the number of children counters a counter splits into is called the *branching_factor* (which is 2^k for a k -bit trie). We can easily see that a RAP tree with *branching_factor* of two is nothing but a binary trie where the path taken to find the counter responsible for an incoming profile event is based on the bits in the event’s identifier. For example, a profile event with identifier $0x10010110$ will be accounted for by a counter which lies in the path with branch labels $\langle 10010110 \rangle$. In Figure 5, the tree on the left shows how the root node acts as a “match-all” node of the RAP tree and how each node splits into two other nodes covering half the range of their predecessor. The tree on the right in Figure 5 shows RAP tree for *branching_factor* of four, in which case the multi-bit trie chooses the path to traverse based on two bits from the identifier at every level. The counter with the smallest range matching the incoming profile event is then incremented.

The distribution of our software implementation of RAP implements the algorithms described in [Mysore et al. 2006]. It also provides initialization and post processing functions to dump the RAP tree for further processing such as those used by optimization tools, or program characterization and visualization tools. Our software version is available at: <http://www.cs.ucsb.edu/~arch/rap>

3.3 Hardware Support for Range Profiling

While a software based approach has many applications, we are interested in using this technique at high speed in run-time systems. For example, we could use this method to analyze a front side bus trace to see what memory is being accessed when a program runs for minutes or hours. We have purposefully designed the algorithms

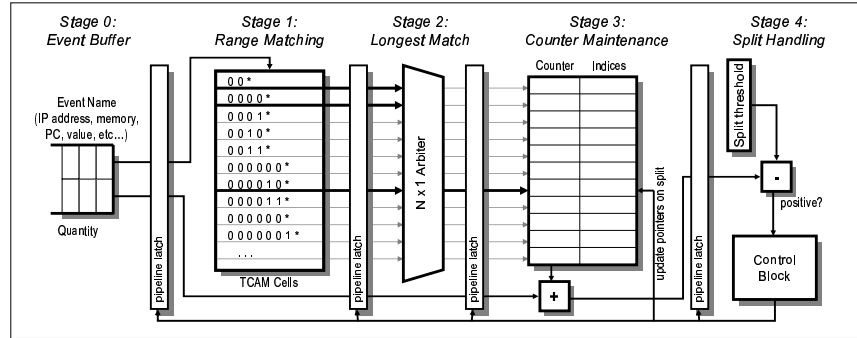


Fig. 6. Architecture for the Pipelined RAP Engine. *Stage0* - Shows the Event Buffer which buffers incoming events and the number of times it is seen since it was previously processed by the RAP engine. *Stage1* - A TCAM range matching provides indices to all TCAM cells which match the incoming event identifier (IP address, PC, value, etc). *Stage2* - A priority Arbiter chooses the longest match by giving it the highest priority. *Stage3* - A set of counters are maintained, each corresponding to an entry in the TCAM array; priority arbiter chooses the counter to be incremented for the incoming event. *Stage4* - For every counter incremented in *Stage3*, a comparator checks the counter value against the current value of the split threshold and if necessary initiates a node split to adapt the precision of the profile maintained.

to allow for an efficient and high speed hardware implementation which can tap into any streaming source of profile information. This source can be from a bus or debug port [fs2] off-chip, or plugged into the back-end of any number of proposed on-chip profiling schemes. We have developed an FPGA prototype of Range Adaptive Profiling that can be interfaced through a high speed network or PCI-X, this is explained in detail in Section 4. In this section we describe a hardware design, and in Section 3.4 we quantify the design in terms of performance, power, and area.

3.3.1 Processor Design. The main features of our conceptual hardware design can be seen in Figure 6. The profiling engine is divided into 5 main stages. In stages 0 and 1, the input events are first buffered, and then all matching ranges are found. In stage 2 the smallest matching range is determined, which then causes the appropriate counters to be updated. Splits and merges are special cases and require pipeline stalls. With the exception of the actual counter increment, each update to the profile tree is independent from the previous. Splits and merges require more work, because they create inter-event dependencies that must be satisfied before more events may be processed. However, compared to updates, splits and merges are very small in number, hence they have little impact on the performance and the total number of stalls is small and bounded.

Stage 0: The small buffer shown at stage 0 in Figure 6 stores incoming points. When the need to perform a merge occurs (periodically and at exponentially decreasing frequency), the pipeline is stalled while the counters are searched for potential merge sets. During this time events will stack up for an estimated ten to a

hundred cycles, and we will need to keep them in a temporary storage so they can be processed later. In the case of a split, the pipeline will need to be flushed and reset to the point directly before where the split should have occurred. In this case the buffer will re-enter those events into the pipeline. It is quite possible to make this buffer pre-process the points by combining *identical* events. We have observed that a 1k buffer can reduce the throughput requirements on RAP by a factor of 10 for code profiling.

Stage 1: For every point fetched from the buffer, we need to find the set of ranges that include that point. This operation is very similar to the Longest Prefix Match and can be carried out in constant time with a Ternary Content Addressable Memory (TCAM) [Pagiamtziz and Sheikholeslami 2004; Agrawal and Sherwood 2006] as shown in Figure 6. Ternary CAM (TCAM) is a special type of CAM [Pagiamtziz and Sheikholeslami 2006] which, due to its special cell design, can store ranges. This is enabled by the fact that TCAM can store don't cares ('*') besides '0' and '1'. It also provides support for searching wildcard bits which can match either '0' or '1'. Hence, wildcard bits can be used in a search operation to indicate that some bits of the search are don't cares and also wildcard bits stored with the data are not used for determining a match. The TCAM sets the appropriate match line high, for all ranges that match. In order to figure out the *smallest range* which is also the *longest prefix*, the TCAM entries have to be partially sorted by prefix length. There can never be matches from two different entries of the same range width. Hence, this stage can be further pipelined by looking at nibble or byte for each comparison [Li et al. 2004].

Stage 2: After the potential matches are identified, we need to find the longest prefix match, which should correspond to the last matching entry. Given N match lines in order, sorted by prefix length, finding the longest match is simply a matter of giving highest priority to longest matches and allowing only one match to proceed. This is exactly the function of a fixed priority $N \times 1$ arbiter. The output of the highest priority line will trigger the word line of the matching counter. Note that while in this paper we assume a TCAM based approach, with a branching factor of b , the tree is really a *multibit trie* and there are a variety of techniques that can be used to build high speed implementations from network algorithms [Srinivasan and Varghese 1999b].

Stage 3: Once the smallest range match has been found, we simply need to update the appropriate counter. To handle a continuous stream of data to the array, one read port and one write port is needed.

Stage 4: The final stage compares the result of the updated counter with the split threshold. If the counter is above the split threshold then the node is expanded to have four children (for *branching_factor*(b) = 4), each initialized to a zero count. The split and merge thresholds are stored in separate registers and recomputed whenever the number of events (n) change. This computation can be done in parallel with other operations as it depends only on n and some predefined values. In our implementation, the split and merge thresholds can be the same, hence just one computation and one register is sufficient. If a split is encountered, the pipeline may need to be flushed to properly account for these new nodes.

In our implementation a split requires making new entries in the TCAM and
ACM Journal Name, Vol. V, No. N, November 2007.

SRAM data array. Four new children nodes are created and inserted in the TCAM with the ranges set appropriately, covering a quarter of the parent range. Corresponding entries in the memory are inserted storing the counter and other information of the newly created nodes. A split node could be either a leaf node or a parent. If the node is a leaf then the split operation involves just setting of a pointer from the parent to the newly created children. If the node is already a parent, but its children do not cover the entire range of the parent (This could be the case after an interior merge as described in Figure 1), then the split also involves an extra operation of identifying the new parent of the existing children and setting the children pointers. In terms of performance, these splits are not a large problem as there can be at most 6400 of them in a given interval, in our implementation.

A merge operation is even more expensive compared to other operations, but by batching them together we reduce the overhead significantly. Batch merges are initiated periodically and in every batch of merges entries in the TCAM are scanned bottom-up to find candidate nodes to be merged. Corresponding SRAM data array entries are then deleted. This recursive operation prunes the RAP tree to provide compacted profile information. If there are 4 billion (2^{32}) events to be profiled, and we assume that there will be at least a thousand (2^{10}) events before we do our first merge, then there will only need to be $32 - 10 = 22$ different batches of merges. Similarly, to profile 2^{64} events, requires $64 - 10 = 54$ batches of merges. If we wish to profile large amounts of data, any cost of doing a merge is quickly amortized.

3.4 Analysis of Required Hardware

In this subsection, we estimate the power consumption, area overhead, and delay of various hardware components involved in the hardware implementation of RAP. We extract and modify the power models from Cacti-3.2 [Shivakumar and Jouppi 0012] and Orion [Wang et al. 2002] tools to model the TCAM, SRAM data array, comparator, priority arbiter, and registers. We then validate our results against some of the published results from high speed circuit design conferences. We assume a very conservative $0.18\mu\text{m}$ technology and we change the voltage supply and various other device parameters accordingly. In particular, we present the worst-case maximum delay and energy consumption by assuming the maximal switching for any particular operation.

Cacti is a widely used cache modeling tool which estimates the area, delay and maximum energy consumption in a cache. Cacti also provides various device parameters which are utilized by Wattch [Brooks et al. 2000] to model power consumption for a CPU model. The Orion [Wang et al. 2002] tool also uses Cacti parameters to measure the power consumption in routers by modeling individual components such as FIFO buffers, crossbars, and arbiters. We make use of all of these tools where appropriate, and validate every estimate that we can.

TCAM - In [Pagiamtziz and Sheikholeslami 2004], a TCAM design of 1024×144 is presented which consumes 2.89fJ/bit/search and the delay is 7 ns in $0.18\mu\text{m}$ CMOS technology. Based on these results from [Pagiamtziz and Sheikholeslami 2004], we estimate the area and energy consumption for a 4096×36 TCAM in $0.18\mu\text{m}$ technology - the area requirement is 19.32 mm^2 and maximum energy consumption is 0.426 nJ for a power-optimized TCAM design.

Priority Arbiter - Orion only provides support for matrix arbiters and round

robin arbiters, which are structurally very different than priority arbiter. We use the result from [Fung and Sachdev 2004; Wang and Huang 2000] to estimate the area, delay and energy consumption of $N \times 1$ priority arbiter. We take 256-bit priority encoder from [Fung and Sachdev 2004] and apply 2-level lookahead to design the 4096-bit priority encoder. We use the result of 256 bit priority encoder, which has a delay of 1.6 *ns*, and energy consumption of 5 *pJ* in 0.6 μm technology. We scale these results and estimate the delay of 4096-bit priority arbiter in 0.18 μm CMOS technology to be 1.16 *ns* and energy consumption to be 63 *pJ*. We estimate the area using the result from [Wang and Huang 2000] in 0.18 μm technology and we find that 4096-bit priority arbiter requires about 0.3 mm^2 of area.

SRAM Data Array - The counters and pointers, as described in the previous sub-section, are stored in a block SRAM. This SRAM data array can be modeled using the Cacti tool by extracting the data-array model of a direct-mapped (DM) cache. We use the data-array component of the direct mapped (DM) cache in which only one wordline is enabled. This data array design is similar to the design of our SRAM data array. We validate the area and energy consumption of 4KB SRAM by comparing it with the results extracted from a memory compiler. A 16KB SRAM in 0.18 μm technology requires an area of about 3.17 mm^2 , takes 1.26 *ns* to access, and consumes 0.54 *nJ*.

Miscellaneous - We also model various other hardware components, which do not significantly contribute to the total area or energy consumption. By modifying Cacti, we can model a comparator by extracting the model used in the set-associative cache. We find that for a 32-bit comparator, the area (0.00034 mm^2) is very insignificant compared to other hardware components and the delay and energy consumption are 0.572 *ns* and 0.0016 *nJ* respectively. Building again off of the device parameters from Cacti, we can model the pipeline registers. For each register, we find that area requirement is 0.0016 mm^2 , maximum delay is 0.164 *ns* and maximum energy consumption is 0.443 *pJ*. To control the TCAM, some additional control registers and data registers are required. We model these registers by the same approach adopted for modeling pipeline registers. For a small TCAM of size 4096 x 36, one control register and two data registers can be used. We model these TCAM registers and find the area and energy consumption to be 0.0026 mm^2 , and 0.613 *pJ* respectively.

We model the data bus wires and control bus wires using the Orion tool. Using 8000 μm wire length, we find the energy consumption for 64-bit bus wires to be 0.94 *nJ*. We estimate the bus wire length from the area and aspect ratio of our hardware design and subsequently, provide the energy consumption result.

Putting it all together - Using 4096 x 36 TCAM and 16KB SRAM data array configurations and summing up the area of all hardware components we find that our Pipelined RAP Engine requires 24.73 mm^2 of area in 0.18 μm technology. The clock frequency is determined by the maximum delay in any pipeline stage and we find that it is governed by TCAM look up stage. The critical path delay in TCAM lookup stage is 7 *ns*. We can aggressively pipeline the TCAM stage by doing byte/nibble comparison at each pipeline stage [Li et al. 2004] and make it comparable to the SRAM stage, which takes 1.26 *ns* time. We also add up the maximum energy components of all the hardware components and we find that a

total of 1.272 nJ energy is consumed. It is also true that an implementation of RAP that can handle 4k different ranges is very aggressive and would most likely be applicable for off-chip profiling, but that for a 400-node version the area and power would be more than a factor of 10 times less. On an average, RAP requires 4 cycles to process an event, and requires 2 cycles each for TCAM and SRAM accesses per event.

4. HARDWARE PROTOTYPING OF RANGE ADAPTIVE PROFILING

To demonstrate on-line profiling with our proposed hardware implementation, we have implemented a prototype for our profiling technique on an FPGA board. The microprocessor communicates with the FPGA board via a PCI bus controller built for an Altera FPGA board (EP1S25F1020C5). This FPGA board has a little over 25,000 logic elements and just under 1 million memory bits which is enough for our implementation. We use Altera Quartus II tool to synthesize our overall design. Next, we describe how we implement each major component including the Queue, TCAM, SRAM, Arithmetic Logic Unit (ALU), and the Controller.

Queue - We implement the *queue* as a synchronous memory module with the following nine ports: *write clock*, *read clock*, *write enable*, *read enable*, *data input*, *data output*, *empty*, *full*, and *reset*. Before the queue can be used, it must be reset to set its internal address pointers to address zero. For simplicity, we implement the queue as positive edge triggered with no negative logic. On the rising edge of a clock, we can read or write, and on the falling edge of a clock we determine if the queue is full or empty. Resetting occurs on the positive edge of the reset signal. For the RAP processor, we implement the queue to hold 1024 values, each 32 bits in size. Our synthesis results show that the *queue* can run at 115 MHz while requiring only 46 logic elements and 32,768 memory bits on our FPGA board.

TCAM - The TCAM is also implemented as a synchronous memory module and it has the following ten ports: *clock*, *write enable*, *read enable*, *search enable*, *select*, *address input*, *data input*, *address output*, *data output*, and *found*. Like the queue, a TCAM must be reset before it can be used, and it is also positive edge triggered with no negative logic. Reading and writing is performed at the positive edge of the clock with their respective enable signals. We search through its memory for the address of a matching data input in one clock cycle. If there are multiple matches, the TCAM will select the match farthest away from address zero, i.e. it will select the match with the largest address. When a search is initiated, on a match, the *found* port produces a '1'; otherwise it is '0'.

In our implementation, the TCAM consists of a decoder module, various memory cell modules, and a priority encoder module. Each memory cell module contains a small amount of synchronous memory and some combinational logic. The synchronous memory contains one address bit and a number of data bits. The select port sends a signal to each memory cell to select between address 0 and address 1. The content of the TCAM memory is encoded in a very specific way; each memory cell breaks up its values in memory such that the upper half of the bits represent a bit mask and the lower half represents a value. For example, if the contents of a memory cell in the TCAM are “00111100”, then the upper half “0011” is the bit mask, and the lower half “1100” is a value; together these translate into 11** for

the TCAM. Another example is, 10110011 which translates to *0**.

For a TCAM match operation, we need to perform the following operations in some simple combinational logic. Let some memory cell in the TCAM contain the value “00111100” (which translates to a TCAM entry of “11**”), and we are trying to match the value “00001110” at the data input port. In this case, we are actually searching for value “1110”, which is the lower half of the value at data input port. The first step is to do *bitwise-or* between the value of the data input with the bit mask of the memory content; we get $0011|1110 = 1111$; we then do the same for the memory contents and we get $0011|1100 = 1111$. The second step is to compare these two values and if they are equal, we have a match. In the above example, we get a match as in both the cases we get the output as “1111”. Similarly, if we search for “1000”, we get the output as $(0011|1000 = 1011)$ and for memory contents we get 1111 and hence we don’t get a match since they are unequal. On multiple matches, the priority encoder finds the match with farthest address from address zero. For a complete search operation, we do the above operation for each memory cell which stores the encoded TCAM entry. Since each memory cell can be read individually and in parallel, the search operation takes one clock cycle. Our synthesis results show that a TCAM with 128 addresses and 64 bit memory cells runs at 51MHz and takes 10,969 logic elements and 16,384 memory bits. The larger the number of memory addresses or data bits, the slower it runs and the more logic elements and memory bits it takes.

SRAM - We implement SRAM as a standard synchronous dual port/dual clock memory module and its seven ports are: *write clock*, *read clock*, *write enable*, *write address*, *data input*, *read address*, and *data output*. We use the standard SRAM with 128 addresses and 60 bits per address. We find that it runs quite fast at 290 MHz taking no logic elements and takes about 7,980 memory bits.

Arithmetic Logic Unit (ALU) - Our ALU implementation is a piece of combinational logic module meant to handle a large amount of data. It keeps track of the following: (1) the maximum number of nodes, (2) current number of nodes, (3) the split and merge thresholds, (4) the merge period, (5) the number of updates, splits, and merges, (6) the number of events processed, (7) the execution count and address being updated, (8) the logarithm of the maximum range in the tree, (9) the error approximation, and (10) the next instruction type, that is, whether the next instruction is an update, split, or merge. When the ALU is reset, most values are initialized to 0, but some values are initialized to 1, namely the current and maximum number of nodes, and yet others are initialized very differently.

Controller - We implement the controller as a state machine with a finite number of states. There are three main states currently operational: (1) Reset, (2) Update, and (3) Split. Each of these main states have sub states, e.g. Reset has 8 total states: Reset, R1, R2,..., R7. Update is unique in that it only has one state; the pipeline handles the rest of the update. The split operation has about 16 states: Split, S1, S2, ..., S15. The first five states are used for flushing the pipeline, while others manipulate reading and writing to and from the TCAM and SRAM.

5. EVALUATING RANGE ADAPTIVE PROFILES

In the sections leading up to this, we have presented the algorithms and designs necessary to perform range adaptive profiling. In this section we analyze the results of our effort by quantifying the memory requirements and errors involved across several SPEC benchmarks, and describe several example use scenarios.

5.1 Profiling with RAP

In trying to examine ranges in the code, values, addresses, or other parameters of a running program, RAP should focus in on the *hot ranges*. A range is considered hot if and only if the total count for that range and all its *non-hot* sub-ranges is above a certain threshold. Note that our definition excludes the possibility that a range is considered hot simply because it has one or more hot children. This is very useful because with a small fixed number of hot ranges we can accurately paint a picture of the distribution of events across the entire range of possible events. For example, when running RAP on a trace of basic blocks, our technique will automatically focus in on the most important regions of code, yet it will provide a balanced overview of the code as a whole. For `gcc` we identify seven distinct regions of the program where each region accounted for more than 10% of the instructions executed.

In addition to code profiles, we also wanted to truly demonstrate the abilities of our scheme by profiling a set of events that has significantly less locality than code profiles. While it has been shown that a single value may account for the top 20% to 40% of all load values, there is a large tail to this distribution which will stress our range profiling system. By building a RAP tree over the set of all values loaded by a program, we can calculate the ranges of values which would cover 50%, 80%, or even 95% of all loads. Figure 7 shows exactly this information for `gzip` and identifies all ranges of load values which are more than 10% hot. In this figure, the hot ranges of load value are shown (with min and max), and they are annotated with their relative weight.

From this figure one can easily see that for `gzip`, load values in the range of $[0, e]$ account for 13.6% of all loads, while the range $[0, fe]$ *excluding* $[0, e]$ accounts for 16.7%. Thus the entire range $[0, fe]$ (including the hot sub-range) accounts for $13.6\% + 16.7\% = 30.3\%$ of loads executed. These summaries are computed completely online and in hardware and could be used to guide optimizations such as value range specialization or to assist in value prediction.

For any profiling system to be feasible, the theoretical and empirical error and memory overheads need to be low. A theoretical analysis of RAP's memory use and error was overviewed in Section 2, and in Sections 5.2 and 5.3 we reevaluate these in the context of code and value profiling. We run our system on a set of programs from the SPEC benchmarks to completion, for reference inputs. The choice of these two types of profiles was governed by factors which can stress-test the RAP system. The locality present in code profiles will stress the upper bounds on memory required for RAP. The heavy tailed distribution of value profiles exercises the range adaptation aspects of RAP. In the rest of this section, we present an analysis of RAP with respect to memory required and error, and illustrate advanced profiling applications of RAP.

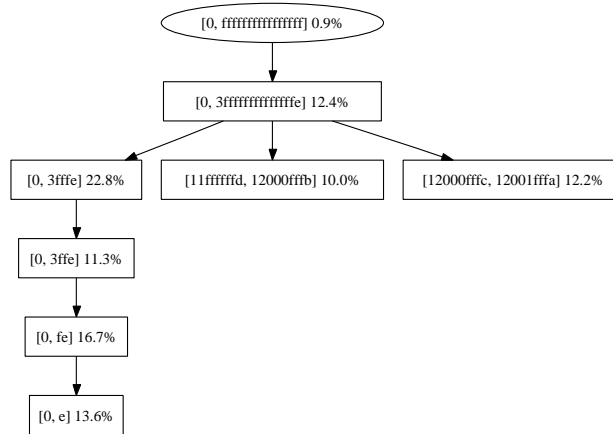


Fig. 7. Hot ranges among the load values in `gzip` as identified by RAP with $\epsilon = 1\%$. We see that there are 7 hot ranges which were encountered for more than 10% of the entire load value stream. Note that this tree is a subset of the RAP tree, showing only the hot nodes

5.2 Memory Analysis

As explained previously, range adaptive profiling stores profiles hierarchically in a tree structure (RAP tree). The number of nodes in the RAP tree will tell us the memory requirement of this scheme. This section gives some practical estimates for various benchmark programs with each node requiring about 128 bits of memory.

Figure 8 shows different benchmarks on the x -axis and the maximum and average number of nodes required by RAP in evaluating these benchmarks is shown on the y -axis. The left hand two graphs show the maximum and average memory required for various benchmarks in identifying hot regions of a code for $\epsilon = 10\%$ (top) and $\epsilon = 1\%$ (bottom). As the tree grows between merge intervals and shrinks after a merge, the maximum memory is the largest of the tree sizes just before the merge operations during the entire run of a benchmark and the average number of nodes indicates the common-case memory requirement. The two graphs on the right present similar parameters for value profiles. We see that a maximum of 500 nodes is sufficient to evaluate code profiles with $\epsilon = 10\%$ for the set of benchmarks. In Section 5.3 we show that with this many nodes we can, on an average, provide 98% accurate information on hot code profiling. We can also observe that `gcc`, which has the highest number of distinct basic blocks, requires a maximum of 453 nodes in the RAP tree for code profiling. The graphs on the right of Figure 8, show similar trends for value profiling. `parser` which has the largest number of load values requires a maximum of 733 nodes and an average of 203 nodes in the RAP tree (Figure 8 for $\epsilon=10\%$). Similarly, the RAP tree requires an average 300 nodes to provide 99% accurate information on load profiles.

An important observation to make is that RAP judiciously allocates counters only if it is sure it is worth allocating them. For example, since the locality among values is less, value profiling with RAP uses less memory (average 300 nodes) compared

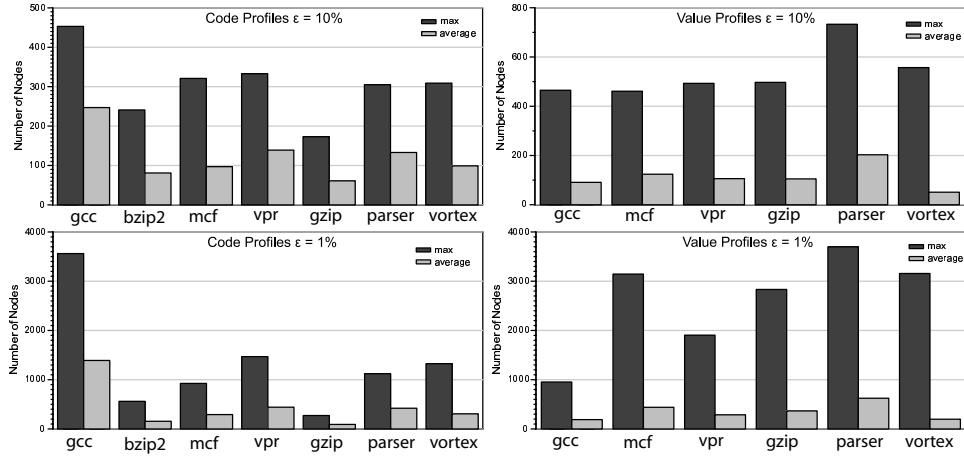


Fig. 8. The number of nodes in the RAP tree which is an indication of the memory required by our profiler is plotted on the y -axis. The left hand two graphs show the maximum and average memory required for various benchmarks in identifying hot regions of a code for $\epsilon = 10\%$ (top) and $\epsilon = 1\%$ (bottom). And the two graphs on the right present similar parameters for value profiles.

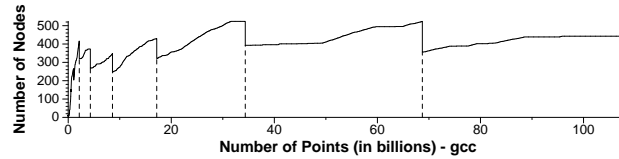


Fig. 9. Number of nodes required to track the basic blocks of gcc with $\epsilon = 10\%$. While the number of nodes is far less than the worst case bounds that we estimated, the pattern of growth (due to splits) and rapid reduction (due to batched merging at points marked by dashed lines) can be clearly seen.

to code profiling (average 450 nodes) which has more locality. This advantage of being able to provide such accurate information using a small amount of memory, is attributed to the splits and merges we do on the RAP tree (as described in Section 2).

Back in Figure 3 we described the bounds on memory requirements as they change over time. To test what happens in a real implementation, we generated Figure 9 which shows the variations of tree size for one such run of gcc. The x -axis represents the number of basic block vectors seen and the y -axis is the number of nodes in the RAP tree. We see a similar pattern to the theoretical expectation, which is the slow building of memory marked by periodic merges which maintain the overall bounds on resource consumption.

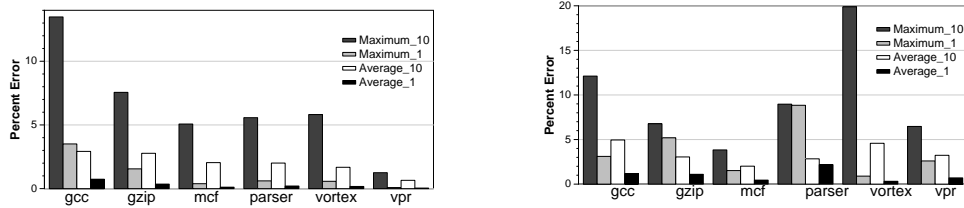


Fig. 10. Percent Error for the hot events identified by RAP for various benchmarks is shown in this figure (for $\epsilon = 1\%$ and 10%). The graph on the left is a measure of accuracy when identifying hot regions of the code. The graph on the right shows similar values for load value analysis.

5.3 Error Evaluation

While the theoretical bounds on error are very useful, if our device is to be used to characterize dynamic program behavior in a real operating environment, the average and worst case percent errors are extremely important.

Due to the way the algorithm is designed, the counts for a range in the tree is always a lower bound on the actual count. Hence, if RAP identifies a node as hot, then that node is guaranteed to be hot. A hot node means that a set of individual events in that range is hot. In cases where the range is a single event, we have identified a hot range with most precision³.

The split threshold is set in such a way that as soon as a node counts events more than a proportion of the total events seen, the node splits into sub-ranges. A merge, similarly, never merges ranges which are hot enough to warrant precise profiling. This ensures that RAP *always profiles with the smallest ranges possible*. Hence, for a given ϵ , we can guarantee that RAP always identifies all hot ranges with the greatest precision possible.

Not only is it important to identify the most frequently observed ranges in a profile stream, but it is equally important to measure how accurately these ranges are quantified. For every hot region identified by RAP,⁴ the estimated counts of the events that contributed to the hot regions were used to compute the percent error. The numbers presented in Figure 10 are a comparison of the estimates that RAP made online, with the actual count that was gathered by making multiple passes through the program's execution, tracking one hot range at a time (as a perfect offline profiler would). Figure 10 shows the percent error in estimating the counts on the hot ranges for each of the different benchmarks. *Maximum_10* and *Maximum_1* is the maximum of the percent errors among all the hot regions for a benchmark, identified in a RAP tree with $\epsilon = 10\%$ and $\epsilon = 1\%$ respectively. Similarly *Average_10* (*Average_1*) is the average of the percent errors for all the

³By precision we mean the ability to zoom into profile ranges as narrow as possible, and by accuracy we refer to error in the quantitative profile information estimated by RAP with respect to a perfect profiler. A perfect profiler is one which can gather event counts with 100% accuracy

⁴For experiments in this section, a region is considered hot if it accounts for more than 10% of the total events. Also, if the hotness threshold is 10%, then the number of hot ranges is at most 10. Similarly for 1% hotness threshold, the maximum number of hot ranges is 100. In our experiments we had a 10% hotness threshold, hence the number of hot ranges were always less than 10.

identified hot ranges within a benchmark with $\epsilon = 10\%$ ($\epsilon = 1\%$). The y -axis in Figure 10 shows percent error for various benchmarks. The graph on the left is a measure of accuracy when identifying hot regions of the code and the graph on the right shows different errors for load value analysis.

In the graph on the left in Figure 10, the benchmark `gcc` shows the highest maximum percent error of 13.5% with $\epsilon = 10\%$. This error of 13.5% was from a hot-range of the code, which was quite narrow and deep in the RAP tree, however, excluding this hot-range, the second maximum percent error in `gcc` is just 3.1%. An important point to draw from this graph is that with $\epsilon = 10\%$, the average percent error is still just about 2%.

Load value analysis, however, was more complex than code profiling because of the wide range of values within which incoming load events could be. With load value analysis (graph on the right in Figure 10), we see that `vortex` has the maximum percent error of around 20% which was due to the hot-value 0 (note, however, that this is still less than 10% error with respect to the entire stream). We also see a negligible percent error with $\epsilon = 1\%$; and with $\epsilon = 10\%$ an average of just 3.4% over all benchmarks. As can be observed, on an average RAP can provide 98% accurate information about code profiles and is 96.6% accurate on value profiles. Trends about program behavior, hot regions, value distribution, memory access patterns are some of the characteristics which can be easily and accurately detected with RAP.

To build a useful and feasible profiler, the error and memory requirements should be bounded absolutely, without reference to the stream length and the type of profile being analyzed. As we have just seen in this subsection, RAP not only precisely identifies range information on a stream of profile events efficiently, but also provides very accurate information.

5.4 Additional Applications of RAP

Thus far we have discussed how RAP can be used to track code and value profiles, and use these to stress-test and evaluate our system. Here we describe several different scenarios where the capabilities of RAP would be useful including: cache-miss value profiling, narrow-width operand profiling, zero-load memory ranges.

Cache-Miss Value Profiling – While we have shown how RAP can be used to profile value locality in a more general sense than simply quantifying “hot values”, architects typically need to target cache misses, rather than simply all loads. Some have hypothesized that while value locality might be present, it may be greatly diminished when only the cache misses are examined. By simply building a RAP tree over the set of all load values which were subject to a cache miss we can quickly quantify this effect. Figure 11 shows the results of performing this analysis averaged over a set of benchmarks. The x -axis shows $\log(\text{range_width})$ of the different hot regions captured by RAP. The y -axis shows the coverage of all events, either loads, DL1 Cache misses, or DL2 Cache misses (depending on the curve). Take for example, DL1 misses. Hot-ranges (those ranges accounting for 10% or more of all DL1 misses) with a size of 2^{16} or less account for about 56% of all DL1 misses. Looking at this figure, it is clear that in fact the value locality of cache misses is *more* than the value locality of all loads.

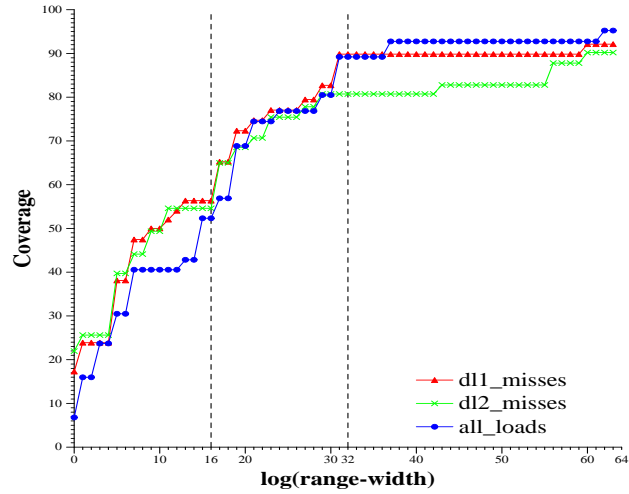


Fig. 11. This figure shows how RAP can be used to extract insightful information about value localities. The x -axis shows the number of bits required to represent the hot ranges of values and y -axis represents the percentage of values profiled

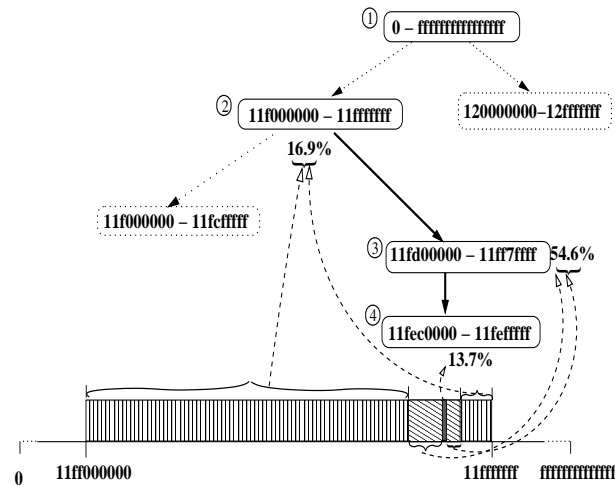


Fig. 12. Memory-value profile characterized by RAP for gcc which identifies from which regions of the memory most zeros are being loaded. The horizontal axis represents the entire memory space. The hot nodes are labeled 1 through 4 and hot regions are shown as pattern-filled boxes. This representation shows a partial RAP tree with hot nodes identified by the solid boxes. The actual RAP tree contains much finer granularity information while this figure shows just those ranges identified by RAP as more than 10% hot

Narrow Operand Profiling – Another application of RAP would be finding regions of code with narrow operands. Finding these regions might benefit operand width prediction and/or bit-width optimized compilation methods. We could build a RAP tree over the set of all instruction PCs which have a narrow operand (for example less than 16 bits). We profiled *gcc* and observed that the narrow-width operations were concentrated in very specific code regions, such as the file `flow.c` which accounted for 38.7% of all narrow-width operations. Within this file, the procedure `propagate_block` accounted for 31%, and a small block in this procedure which processed the live registers accounted for 6.4%.

Zero-load Memory Ranges – A different but related type of profile is to find out which regions of the data memory are responsible for load of a particular value, for example zero. This memory-value profiling could be used to guide bus compression schemes or track potentially inefficient data structures. Figure 12 shows a RAP tree for *gcc* built over the set of all memory addresses from which a zero was loaded. If the optimizers goal was to reduce the number of zero-loads, these memory ranges would be the best place to target. The horizontal axis represents the entire range of data memory (0-ffffffffffff). We focus on the hot nodes identified by RAP (labeled 1-4). We have zoomed in to show how RAP precisely identified distinct ranges which accounted for 16.9% (Node 2), 54.6% (Node 3) and 13.7% (Node 4) of the zero loads. For example, the address ranging from 11fd00000-11ff7ffff (Node 3) accounts for a total of $13.7\% + 54.6\% = 68.3\%$ of all zero loads in *gcc*. In fact, it was also observed that any load to this region has about 38% percent chance of being a zero.

In general, any event (cache misses, 0-loads, exceptions, ...) can be mapped using RAP, to the code that caused them, the memory address that was referred to, or the value on which an instruction operated. While the above profiling scenarios are not complete optimizations, they provide evidence that RAP has the potential to be both general purpose across many different types of profiles, and powerful enough to encourage new types of profiling.

5.5 Multi-Dimensional RAP

Among the thousands of micro-architectural events that can be observed during a program execution, the ability to profile a couple of the most important ones (code profiles, value profiles, memory access profiles, etc.) would be very useful. Optimizations, often rely not just on inferences drawn from a single profile stream but on a combination of inferences drawn from multiple profile streams, even better from a correlation of these different streams. For example, it may be very useful to understand which regions of the code are hot *and* what are the hot values loaded within this code region, or what are the most frequently accessed memory locations within this code region.

While simultaneous profiling of multiple profile streams with RAP helps point out the hot spots in each stream independently of each other, with Multi-dimensional RAP, we introduce the capability to profile different profile streams while still maintaining the correlation among these streams. At the heart of multi-dimensional RAP is the same algorithm as explained in Section 2 except that the nodes in the RAP tree now store N -dimensional ranges (two in the case of 2-D RAP tree) instead of a one dimensional range. To demonstrate the usefulness of multi-dimensional

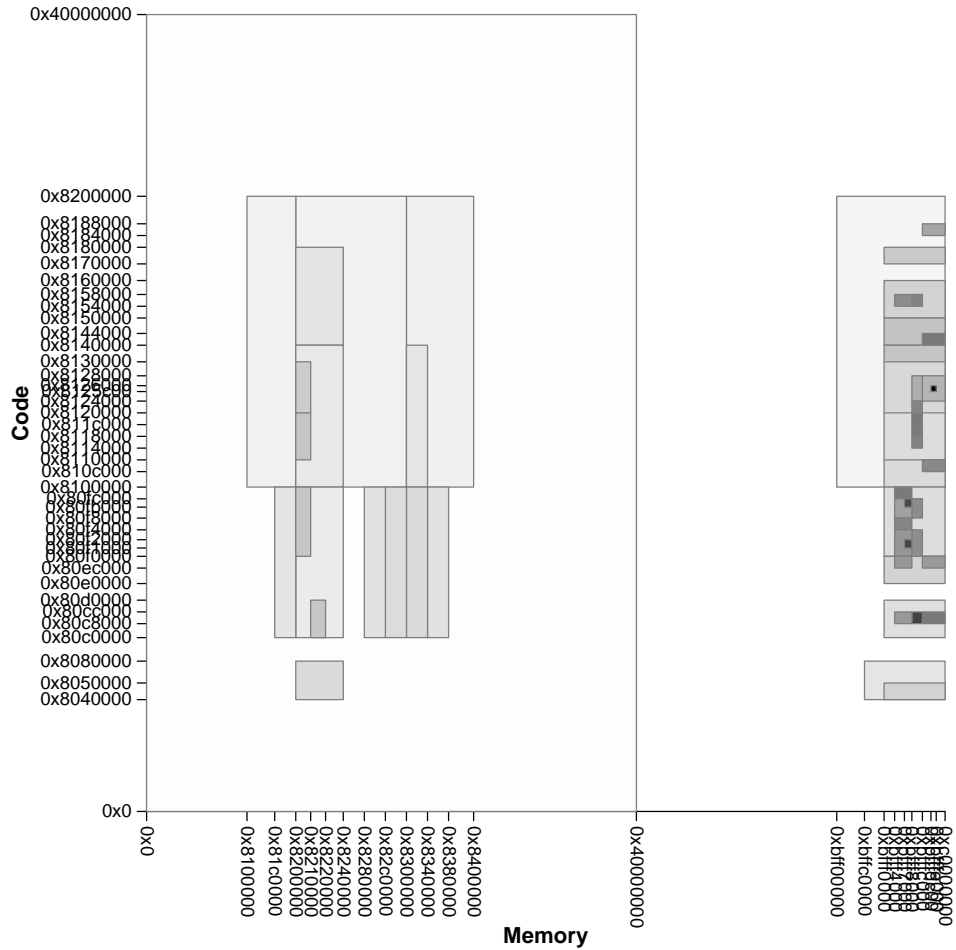


Fig. 13. Multi-dimensional RAP profile for gcc identifying all the hot code regions (represented on the y -axis) and the hot memory addresses accessed (represented on the x -axis) by a particular code region. In this figure, the hot regions are represented by the rectangles and the darker the color of a region, the hotter it is.

RAP, we show a simple example of a 2-D profiling of gcc in Figure 13. In this case, every node in the RAP tree stores two ranges, one for the code region, and one for the memory region accessed within that code region. The code region is represented on the y -axis and the memory accessed is represented on the x -axis. For example, the bottom left hand rectangle identifies a hot code region as $[0x804000, 0x808000]$ and the hot memory regions accessed by this region as $[0x8200000, 0x8240000]$. The shaded rectangles represent the hot regions identified by RAP, the darker the shade of a rectangle, the hotter it is. In general, multi-dimensional RAP can be applied to obtain summary information about the correlation between different profile streams and their hot regions.

6. RELATED WORK

Range Adaptive Profiling is a novel method to provide hierarchical summary information on a stream of events. While we present a hardware based framework for dealing with vast amounts of profiling data, our technique builds on the profiling work of many other researchers. In this section we briefly summarize some of this work and relate it to our own contributions. We classify our related work into two broad categories:

Software Based Profiling - Software systems can be either statically instrumented with instrumentation tools such as ATOM [Srivastava and Eustace 1994] or dynamically through just-in-time compilers [Kral 1998]. In software profiling, most of the effort has been spent on reducing the performance overhead of instrumentation such as through sampling [Arnold and Ryder 2001] or bursty tracing [Hirzel and Chilimbi 2001]. Dynamic hot-path prediction techniques are described in [Duesterwald and Bala 2000]. Value profiles are another important form of profiles [Calder et al. 1997], which identify value invariance and proposes optimizations through Convergent Profiling. There is software work on sampling more intelligently and even on compressing trace information to reduce the overheads involved. Larus [Larus 1999] provides a technique to capture, in a compressed form, a program's dynamic control flow. The idea of using software to extract a hierarchy of information using grammars, has been used to implement efficient data prefetching mechanisms [Chilimbi 2001; Chilimbi and Hirzel 2002]. A general purpose software framework for dealing with compressed profile data is proposed in [Zhang and Gupta 2004]. While these are powerful software mechanisms, they are not directly applicable to the problem of managing a very small number of hardware counters to enable high-throughput hardware-only profiling.

Hardware Assisted Profiling - The current industrial practice in hardware performance monitoring is performance counters, and several modern machines now support this idea [Corporation 1995; 1997; Inc 1995]. These simple counter based schemes, while useful, suffer from a lack of flexibility and require significant software management in order to extract useful information [Anderson et al. 1997]. Many researchers have examined the next steps that hardware assisted profiling should take. Proposed schemes range from those which use existing hardware on the processor to gather information which is later processed by a software program [Anderson et al. 1997; Conte et al. 1994; Peri et al. 1999], to programmable profiling co-processors [Zilles and Sohi 2001]. [Conte et al. 1996] uses profile buffers to collect and analyze information. Sastry et. al. in [Sastry et al. 2001] provide a framework for designing a variety of stream compressors and propose the stratified sampling scheme. An extension of the stratified sampling scheme is proposed by [Narayanasamy et al. 2003] which aims at reducing the cost of delivering gathered profile and proposes multi-hash and interval based profiling. Though these schemes provide efficient ways to process data they are not flexible enough to accommodate general queries. ProfileMe [Dean et al. 1997] and Relational Profiling Architecture are flexible and versatile schemes for gathering profile information. Zilles and Sohi [Zilles and Sohi 2001] in their co-processor approach, design hardware to analyze the stream and compress it to provide concise and distilled profile information to the main processor. It has the ability to consider only a subset of the instruc-

tions for profiling and refocus resources after an instruction has been sufficiently characterized.

Our approach is orthogonal to most of the above approaches because RAP concentrates on building a useful online summary of the data, no matter what method is used to gather the data. RAP can be completely implemented in hardware and has the ability to efficiently identify the most important ranges of the profile and provide accurate information on the entire profile with very low overheads. We believe that there are important similarities between profiling a program executing billions of instructions per second and trying to monitor and analyze high speed networks [Hershberger et al. 2004; Estan et al. 2003; Kruegel et al. 2002] and that there is potential for further research along these lines. Indeed, RAP has been designed to be adaptable to a variety of different data streams that need to be processed at very high speed, and may even be applied in analyzing network traffic.

7. CONCLUSIONS

Amdahl's law shows us that the common case is most important to performance so it makes sense to bias allocated resources towards the common case. The problem is that the common case changes as the program executes and we end up with a chicken-and-egg type of problem. In this paper we present Range Adaptive Profiling - a novel scheme to efficiently, adaptively and intelligently summarize high bandwidth streams of profile data. It allows users to specify a parameter (ϵ) which bounds the error with respect to the size of the input stream and also provides guarantees on worst case memory bounds independent of the size of the input stream *in a fully streaming fashion* (with only one-pass). This method can be applied to software profiling, and with the use of a specialized pipelined architecture, can be accelerated with hardware.

While it is not yet clear whether Range Adaptive Profiling will be general purpose enough to cover all profile types of interest, we have shown that it can make sense for summarizing at least three profile types: load values of cache misses, instruction PCs of narrow width operands, and memory addresses of zero-loads. The applicability of RAP can be further extended with multi-dimensional profiling which allows adaptive ranges over two or more variables. With this extension it is possible to handle edge profiles, data-code correlation studies, and general tuple space profiles, the details of which are beyond the scope of this paper. It may further be possible to unify our proposed techniques with existing sampling based schemes to create a single general purpose profiling system. While this future work may prove fruitful, to guide our initial algorithm and hardware design we have used load values and code profiling to measure the overheads of RAP and also to show the versatility of the scheme.

While some have shown the frequency of the top 50 individual loaded values in a program which might cover 40% of the program, our technique can *automatically generate range summaries* which include every value loaded in an entire SPEC benchmark, and we believe this type of analysis to be the first of its kind. This information could be used to drive many run-time optimizations including code specialization, value prediction, and bus encoding. While RAP has good worst case bounds, in the common case it is even better. For a set of benchmark programs from

SPEC, we can provide 98% accurate information about hot code regions with only 8k bytes of memory and 99.73% accurate information with 64k bytes of memory. The RAP method is suitable for intelligent processing of the many different profile streams that may be generated from either a processor or computer network, and our future work will extend this technique to handle new forms of profiling in the processor.

Acknowledgments

The authors would like to thank Cliff Young, Prof. Rajiv Gupta, and the anonymous reviewers for their helpful feedback. This research was funded in part by National Science Foundation Grants CCF-0514738, CCF-0702798, and NSF Career Grant CCF-0448654.

REFERENCES

- AGRAWAL, B. AND SHERWOOD, T. 2006. Modeling tcam power for next generation network devices. In *Proc. of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. Austin, TX.
- ANDERSON, J., WEIHL, W., BERG, L., DEAN, J., GHEMAWAT, S., HENZIGER, M., LEUNG, S., SITES, R., VANDEVOORDE, M., AND WALDSPURGER, C. 1997. Continuous Profiling: Where Have All the Cycles Gone? *ACM Transactions on Computer Systems (TOCS)* 15, 4 (November), 357–390.
- ARNOLD, M. AND RYDER, B. 2001. A Framework for Reducing the Cost of Instrumented Code. In *SIGPLAN Conference on Programming Language Design and Implementation*. 168–179.
- BALL, T. AND LARUS, J. R. 1996. Efficient Path Profiling. In *International Symposium on Microarchitecture*. 46–57.
- BROOKS, D. AND MARTONOSI, M. 1999. Dynamically Exploiting Narrow Width Operands to Improve Processor Power and Performance. In *Proceedings of the The Fifth International Symposium on High Performance Computer Architecture*. IEEE Computer Society, Washington, DC, USA, 13.
- BROOKS, D., TIWARI, V., AND MARTONOSI, M. 2000. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. In *ISCA*. 83–94.
- BUCK, B. AND HOLLINGSWORTH, J. K. 2000. An API for Runtime Code Patching. *The International Journal of High Performance Computing Applications* 14, 4, 317–329.
- BUS, B. D., CHANET, D., SUTTER, B. D., PUT, L. V., AND BOSSCHERE, K. D. 2004. The design and implementation of fit: a flexible instrumentation toolkit. In *PASTE '04: Proceedings of the 5th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. ACM Press, New York, NY, USA, 29–34.
- CALDER, B., FELLER, P., AND EUSTACE, A. 1997. Value Profiling. In *International Symposium on Microarchitecture*. 259–269.
- CHILIMBI, T. 2001. Efficient Representations and Abstractions for Quantifying and Exploiting Data Reference Locality. In *Conference on Programming Languages Design and Implementation*.
- CHILIMBI, T. AND HIRZEL, M. 2002. Dynamic Hot Data Stream Prefetching for General-Purpose Programs. In *Conference on Programming Languages Design and Implementation*.
- CONTE, T. M., MENEZES, K. N., AND HIRSCH, M. A. 1996. Accurate and Practical Profile-driven Compilation Using the Profile Buffer. In *International Symposium on Microarchitecture*. 36–45.
- CONTE, T. M., PATEL, B. A., AND COX, J. S. 1994. Using Branch Handling Hardware to Support Profile-driven Optimization. In *International symposium on Microarchitecture*. 12–21.
- CORPORATION, D. E. 1995. Alpha 21164 Microprocessor Hardware Reference Manual.
- CORPORATION, I. 1997. Pentium(r) Pro Processor Developer's Manual. In *McGraw-Hill*.
- DEAN, J., HICKS, J., WALDSPURGER, C., WEIHL, W., AND CHRYSOS, G. 1997. ProfileMe: Hardware Support for Instruction-Level Profiling on Out-of-Order Processors. In *International Symposium on Microarchitecture*. 292–302.

- DUESTERWALD, E. AND BALA, V. 2000. Software Profiling for Hot Path Prediction: Less is More. In *Architectural support for programming languages and operating systems*. 202–211.
- ESTAN, C., SAVAGE, S., AND VARGHESE, G. 2003. Automatically Inferring Patterns of Resource Consumption in Network Traffic. In *SIGCOMM '03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*. Karlsruhe, Germany.
- fs2. First Silicon Solutions. Home page. <http://www.fs2.com>.
- FUNG, W. W. AND SACHDEV, M. 2004. High Performance Priority Encoder for Content Addressable Memories. In *Micronet R&D Annual Workshop*.
- GUPTA, R., BERSON, D., AND FANG, J. 1998. Path profile guided partialredundancy elimination using speculation. In *IEEE International Conference on Computer Languages (ICCL)*. 230–239.
- HEIL, T. AND SMITH, J. E. 2000. Relational Profiling: Enabling Thread-level Parallelism in Virtual Machines. In *International symposium on Microarchitecture*. 281–290.
- HERSHBERGER, J., SHRIVASTAVA, N., SURI, S., AND TOTH, C. 2004. Adaptive Spatial Partitioning for Multidimensional Data Streams. In *International Symposium on Algorithms and Computation (ISAAC)*.
- HEWLETT-PACKARD. 1994. PA-RISC 1.1 Architecture and Instruction Set Reference Manual.
- HIRZEL, M. AND CHILIMBI, T. 2001. Bursty Tracing: A Framework for Low-overhead Temporal Profiling. In *4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*.
- INC, M. T. 1995. Mips R10000 Microprocessor User's Manual.
- JACOBSON, Q., ROTENBERG, E., AND SMITH, J. 1997. Path-based next trace prediction. In *30th IEEE/ACM International Symposium on Microarchitecture (MICRO)*.
- KLEINOSOWSKI, A. AND LILJA, D. J. 2002. Minnespec: A new spec benchmark workload for simulation-based computer architecture research. In *Computer Architecture Letters*. Vol. 1.
- KRALL, A. 1998. Efficient JavaVM Just-in-Time Compilation. In *International Conference on Parallel Architectures and Compilation Techniques*. 205–212.
- KRUEGEL, C., VALEUR, F., VIGNA, G., AND KEMMERER, R. 2002. Stateful Intrusion Detection for High-Speed Networks. In *IEEE Symposium on Security and Privacy*. 285–293.
- LAMPSON, B., SRINIVASAN, V., AND VARGHESE, G. 1999. Ip lookups using multiway and multicolumn search. *IEEE/ACM Trans. Netw.* 7, 3, 324–334.
- LARUS, J. 1999. Whole program paths. In *Conference on Programming Languages Design and Implementation (PLDI)*. 259–269.
- LI, X., LIU, Z., LI, W., AND LIU, B. 2004. SCP-TCAM: A Power-Efficient Search Engine for fast IP Lookup. In *ISBN Proceedings*.
- LIPASTI, M. H. AND SHEN, J. P. 1996. Exceeding the dataflow limit via value prediction. In *29th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 226–237.
- LOH, G. H. 2002. Exploiting data-width locality to increase superscalar execution bandwidth. In *MICRO 35: Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*. IEEE Computer Society Press, Los Alamitos, CA, USA, 395–405.
- LUK, C. AND ET. AL. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. In *Conference on Programming Languages Design and Implementation (PLDI)*.
- MYSORE, S., AGRAWAL, B., SHERWOOD, T., SHRIVASTAVA, N., AND SURI, S. 2006. Profiling over Adaptive Ranges. In *Proceedings of the Fourth International Symposium on Code Generation and Optimization (CGO-4)*. IEEE Computer Society, New York, NY, USA, 147–158.
- NARAYANASAMY, S., SHERWOOD, T., SAIR, S., CALDER, B., AND VARGHESE, G. 2003. Catching Accurate Profiles in Hardware. In *Int. Symp. on High-Performance Computer Architecture*. 269–280.
- PAGIAMTZIS, K. AND SHEIKHOESLAMI, A. 2006. Content-addressable memory (cam) circuits and architectures: A tutorial and survey. In *IEEE Journal of Solid-State Circuits*. Vol. 41.
- PAGIAMTZIS, K. AND SHEIKHOESLAMI, A. 2004. A Low-Power Content-Addressable Memory (CAM) Using Pipelined Hierarchical Search Engine. *IEEE Journal of Solid-State Circuits*.
- ACM Journal Name, Vol. V, No. N, November 2007.

- PERI, R. V., JINTURKAR, S., AND FAJARDO, L. 1999. A Novel Technique for Profiling Programs in Embedded Systems. In *ACM Workshop on Feedback-Directed and Dynamic Optimization*.
- RUBIN, S., BODIK, R., AND CHILIMBI, T. 2002. An efficient profile-analysis framework for data layout optimizations. In *The 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*.
- SANCHEZ, M., BIERSACK, E., AND DABBOUS, W. 2001. Survey and taxonomy of IP address lookup algorithms. *IEEE Network Magazine* 15, 2, 8–23.
- SASTRY, S., BODÍK, R., AND SMITH, J. 2001. Rapid Profiling via Stratified Sampling. In *Annual International Symposium on Computer Architecture*. 278–289.
- SHIVAKUMAR, P. AND JOUPPI, N. P. 2001/2. Cacti 3.0: An Integrated Cache Timing, Power and Area Model.
- SRINIVASAN, V. AND VARGHESE, G. 1999a. Fast address lookups using controlled prefix expansion. *ACM Transactions on Computer Systems* 7, 1 (Feb.), 1–40.
- SRINIVASAN, V. AND VARGHESE, G. 1999b. Fast Address Lookups Using Controlled Prefix Expansion. *ACM Transactions on Computer Systems* 7, 1 (Feb.), 1–40.
- SRIVASTAVA, A., EDWARDS, A., AND VO., H. 2001. Vulcan: Binary transformation in a distributed environment. technical report msr-tr-2001-50, microsoft research.
- SRIVASTAVA, A. AND EUSTACE, A. 1994. ATOM: A System for Building Customized Program Analysis Tools. In *Conference on Programming Languages Design and Implementation*. 196–205.
- WANG, H., ZHU, X., PEH, L., AND MALIK, S. 2002. Orion: A Power-Performance Simulator for Interconnection Networks. In *Proceedings of the 35th International Symposium on Microarchitecture*. Istanbul, Turkey.
- WANG, J. AND HUANG, C. 2000. High-Speed and Low-Power CMOS Priority Encoders. *IEEE Journal of Solid-State Circuits* 35, 10 (October), 1511–1514.
- YANG, J. AND GUPTA, R. 2002. Frequent Value Locality and its Applications. In *ACM Transactions on Embedded Computing Systems*.
- YANG, J., ZHANG, Y., AND GUPTA, R. 2000. Frequent value compression in data caches. In *International Symposium on Microarchitecture*. 258–265.
- YOUNG, C. AND SMITH, M. 1998. Better global scheduling using path profiles. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 115–123.
- ZHANG, X. AND GUPTA, R. 2004. Whole execution traces. In *Proceedings of the 37th International Symposium on Microarchitecture*. IEEE Computer Society, Washington, DC, USA, 105–116.
- ZHANG, Y. AND GUPTA, R. 2001. Timestamped whole program path representation and its applications. In *Conference on Programming Languages Design and Implementation (PLDI)*. 180–190.
- ZHOU, H., FLANAGAN, J., AND CONTE, T. M. 2003. Detecting global stride locality in value streams. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*. ACM Press, New York, NY, USA, 324–335.
- ZHOU, P., QIN, F., LIU, W., ZHOU, Y., AND TORRELLAS, J. 2004. iWatcher: Efficient Architectural Support for Software Debugging. In *Proceedings of the 31st annual International Symposium on Computer Architecture (ISCA '04)*.
- ZILLES, C. AND SOHI, G. 2001. A Programmable Co-Processor for Profiling. In *The Seventh International Symposium on High Performance Computer Architecture*.