A Fast On-Chip Profiler Memory using a Pipelined Binary Tree

Roman Lysecky, Susan Cotterell, Frank Vahid*
Department of Computer Science and Engineering
University of California, Riverside
{rlysecky, susanc, vahid}@cs.ucr.edu
*Also with the Center for Embedded Computer Systems at UC Irvine

Abstract

We introduce a novel memory architecture that can count the occurrences of patterns on a system's bus, a task known as profiling. Such profiling can serve a variety of purposes, like detecting a microprocessor's software hot spots or frequently used data values, which can be used to optimize various aspects of the system. The memory, which we call ProMem, is based on a pipelined binary search tree structure, yielding several beneficial features, including non-intrusiveness, accurate counts, excellent size and power efficiency, very fast access times, and the use of standard memories with only simple additional logic. The main limitation is that the set of potential patterns must be preloaded into the memory. We describe the ProMem architecture, and show excellent size and performance advantages compared with CAM (content-addressable memory) based designs.

Keywords

Profiling, binary search tree, pipelined binary search tree, on-chip profiler, memory, design, high-performance.

1. Introduction

Counting the frequency of occurrence of patterns on a computer bus is a task that can be useful in a wide variety of problems. One common use is to perform software profiling. In the domain of computing, *profiling* generally means to determine the relative frequency of code regions of interest as a program executes, ranging from fine-grained items like individual statements or basic blocks, to coarser-grained items such as loops or subroutines. The term profiling has also been used to refer to determining the relative frequencies of values that a variable takes on during program execution. However, designing hardware that profiles an executing application non-invasively yet accurately is non-trivial, since updating profile counts fast enough is hard.

Most previous profiling approaches, being intended for desktop computing systems, introduce runtime overhead. In particular, either they "instrument" the code by inserting additional code into the application binary [4], or they interrupt the processor at particular intervals to sample the processor's registers [1][3]. However, for embedded systems, runtime overhead is often not acceptable, since very tight real-time constraints must be met.

A hardware technology that might seem to solve the problem of detecting patterns in large pattern sets is content-addressable memories (CAM). CAMs provide fast searches for a key in a large data set. Given the data, a CAM returns the address at which the key (data) resides in a memory. One type of CAM uses a fully associative memory, which simultaneously compares every location with the key. Fully associative memories have some drawbacks for large data sets. First, their access time increases as they become larger, since the signals for the key must be distributed to all memory locations simultaneously. Thus, larger CAMs have slower access times. Second, their size is greater than regular RAM (random access memory). A typical SRAM (static RAM) cell uses six transistors, whereas a typical fully associative CAM cell having built-in comparison logic requires ten transistors. Third, such a CAM may consume excessive power, since every word of the CAM will be active on each access.

We therefore have developed a memory architecture specifically intended for profiling. The profiler memory can be used to keep counts of hundreds or thousands of bus patterns simultaneously, in contrast to previous profiler hardware. Yet, the memory has a simple interface to the bus it is monitoring, is very size efficient, and can be composed from a collection of standard register files or memories surrounded by a small amount of additional logic. By using a novel pipelined binary search tree architecture, our profiler memory achieves single-cycle throughput, meaning complete accuracy even when monitoring very fast sequences of patterns, such as those on the address bus of modern embedded processors that fetch one instruction per cycle. The binary search tree is implemented with a separate module for each tree level, and thus scales well, becoming even more efficient the larger it gets.

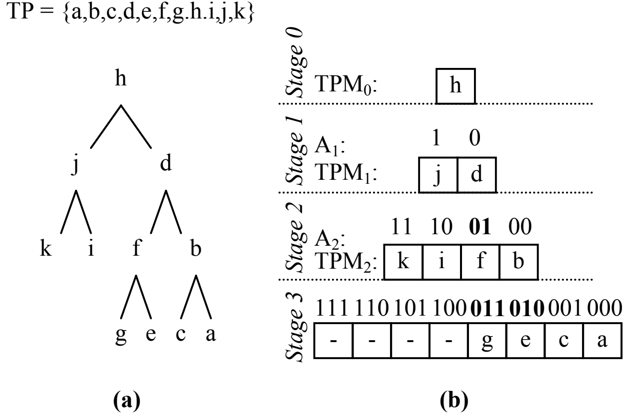
2. Problem Definition

Although we present the memory in the context of performing code profiling by monitoring a processor's address, the memory's use may actually extend to a variety of domains, such as network monitoring, or any of a wide range of scientific applications (chemistry, biology, physics) in which specific patterns in a stream of data must be counted – applications that could range from chemical analysis to DNA matching to space monitoring.

Code profiling requires that we count the frequency of target addresses appearing on a microprocessor's address bus, during some given period. We assume that the set of target addresses has been given to us. They could correspond to statements, blocks, loops, subroutines, variables, arrays, or any combination thereof. Our approach is independent of how these addresses are selected, what they represent, or how they will be used.

We assume the address bus being monitored contains the actual addresses of interest. We also assume that virtual memory is not being used (the common case in embedded systems), so that the physical addresses on the address bus need not be further translated to virtual addresses. We assume target addresses could appear as frequently as every clock cycle. We assume the profiling circuit operates at the same (or lower) clock frequency than the bus being monitored – specifically, we do not assume the luxury of the profiling circuit having a faster clock than the bus.

Figure 1: Conceptual view of ProMem: (a) binary search tree for target pattern set, (b) storing each level (stage) in target pattern memories (TPM) with corresponding memory addresses *A* shown.



We state the problem generally as follows. We must monitor a bus *B*, having a width *w*, for a period of *M* clock cycles. During each cycle *t*, a pattern p_t appears on *B*, thus forming an input pattern set $P=\{p_1, p_2, ... p_M\}$. A pattern is simply a combination of *w* 0s and 1s on the bus. We are given a target pattern set $TP=\{tp_1, tp_2, ... tp_n\}$; each target pattern is a combination of *w* 0s and 1s. Our goal is to maintain a set of target pattern counts $C=\{ctp_1, ctp_2, ... ctp_n\}$, such that ctp_i equals the numbers of times pattern tp_i was seen on *B* during the period *M*. Specifically:

$$ctp_i = \sum_{t=1}^{M} \begin{Bmatrix} 1, & if \ p_t = tp_i \\ 0, & otherwise \end{Bmatrix}$$

We assume our method of monitoring and counting must be non-intrusive. In other words, we cannot introduce extra clock cycles or change the patterns occurring over bus *B*. Non-intrusiveness is important in embedded systems, since hard real-time constraints often must be met, and precise timing of input/output operations is often imperative. Increasing runtime by even the smallest amount can lead to constraint violations, radically different system behavior, and/or system failure.

We would also like our memory to be size and power efficient, and to utilize standard memories generated by memory compilers like those from Artisan [2].

3. A Self-Profiler Memory Architecture

The key to building an efficient self-profiler memory is to recognize that we do not need single cycle lookup, or even single-cycle write, but rather just single-cycle update throughput. In other words, the memory must be able to accept a new pattern every cycle, but the memory need not actually update the count field of a matching target pattern until many cycles later.

3.1 Pipelined Binary Search Tree

To achieve a throughput of one new pattern every cycle, we implement our ProMem memory structure using a pipelined binary search tree structure to find the location of the given input pattern and update associated pattern count. Figure 1(a) shows a binary search tree for a target pattern set *TP*={a, b, c, d, e, f, g, h, i, j, k}. We will implement the tree by using a separate module for each tree level, with each module implementing a pipeline stage.

Figure 2: ProMem Module Design: (a) design of a module for stage *s* (> 0), (b) structure of ProMem using modules (each stage is actually twice the size of the previous stage).

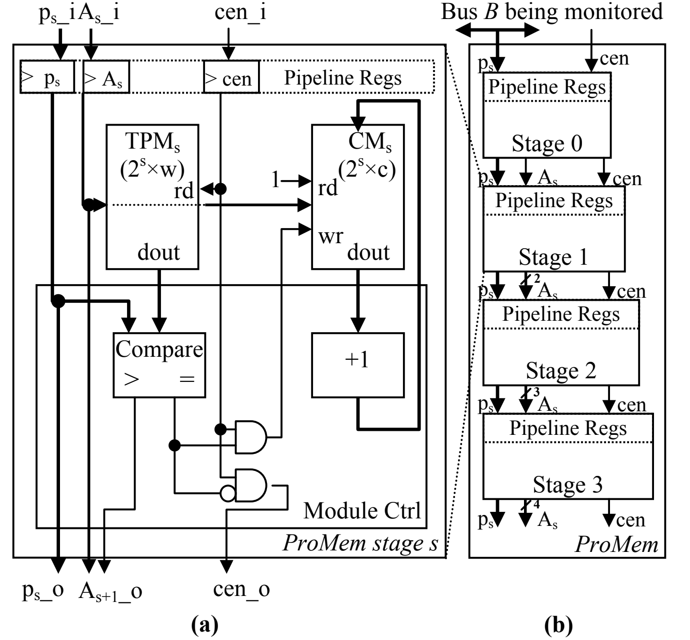


Figure 1(b) illustrates the target patterns that would appear in a memory (target pattern memory, or TPM) in each module, assuming a four stage ProMem design. We place the target patterns in memory such that the following property holds:

> *A node's children have the same high-order address bits as the parent's address, with the low order address bit of each child indicating left (1) or right (0) child.*

This relationship can be observed, for example, in the bolded bits of Figure 1(b): *f* has address 01 in TPM_2, and has left child *g* and right child *e* with addresses 011 and 010 in TPM_3.

3.2 Memory Architecture

Figure 2(a) shows the design of a single module within ProMem. Each level's module consists of registers to latch the input pattern, address, and enable signal (collectively referred to as the Pipeline Register), a memory containing the target patterns (TPM), a memory containing target patterns' corresponding counts (CM), a comparator, an incrementer, and some required logic.

From the pipeline register, the input pattern will be compared against the contents of the TPM_s at address A_s_i. The comparator used in the comparison has two outputs: a greater than comparison and an equal to comparison. If the current stage is enabled by the *cen_i* signal and the input pattern is equal to the target pattern, the *cen_o* signal will be set to 0, indicating the pattern was found and no further searching is required. In addition, the write port of the CM will be enabled to increment the associated count value. However, because this requires a read and write of the same memory location, we will need to use a memory that has an independent read port and an independent write port. Alternatively, if the input pattern does not match the current target pattern, the result from the greater than comparison will be concatenated with the input address, A_s_i, to create the address for the next stage, A_{s+1}_o, and *cen_o* will be set to 1,

Table 1: ProMem area and timing results (8 stages, 32-bit target patterns).

Stage	TPMs + CMs (Gates)	Pipeline Register (Gates)	Module Controller (Gates)	Module Controller (% total)	Total (Gates)	Access Time (ns)
0	774	268	640	38%	1682	4
1	1618	274	713	27%	2606	4
2	3156	284	753	18%	4193	4
3	6468	298	812	11%	7577	4
4	12925	307	823	6%	14054	4
5	25870	313	934	3%	27117	4
6	53254	322	995	2%	54572	4
7	115477	327	1287	1%	117090	4
Total	**219541**	**2393**	**6957**	**3%**	**228891**	**4**

indicating we have not found the target pattern and need to continue searching.

To deal with the issue of locations within our binary search that do not contain valid addresses, we implement a simple scheme that extends the width of the TPM by one bit. When a valid target pattern is written to the TPM, the pattern is concatenated with a single bit having the value of 1. Then, to compare the target pattern to the input pattern, we also concatenate the input pattern with a 1 bit before the pattern reaches the comparator. If the two values when compared are equal, we have found the target pattern. Otherwise, to determine if the current target pattern is valid, we check the most significant bit of the current target pattern. If the bit is 0, this indicates that we do not have a valid target pattern, in which case we stop searching.

To construct the entire ProMem structure, we simply connect a module to the module for the next stage, as shown in Figure 2(b). Thus, to achieve a ProMem design that can handle a target pattern set with 1023 entries, we will need to create a binary search tree structure with 10 stages. However, using the modular design for each level, extending the number of target patterns is as simple as adding another level to the design.

We also implement a mechanism that utilizes the existing pipeline structure for writing the target patterns into ProMem and reading out the counts after monitoring is complete. However, using this implementation, the entries written to ProMem must be in sorted order. A description of how writing to and reading from ProMem is implemented can be found in [5].

4. Results

We implemented ProMem in VHDL. We designed ProMem as a combination of stages that are connected together in a top-level entity. Because stage 0 of our ProMem has a different structure then the remaining stages, stage 0 was implemented as its own entity. The remaining stages use a single entity that contains a generic *STAGE*, which specifies the stage of the instantiated entity. Therefore, adding another stage to the design is as simple as instantiating another stage and connecting the outputs of the previous stage to the inputs of the new stage. We tested our VHDL code at structural RTL level as well as the gate-level description generated from synthesis.

Table 1 shows the area results from synthesizing an 8-stage ProMem using 32-bit target patterns. We synthesized ProMem with Synopsys Design Compiler using the UMC 0.18 technology and memory libraries provided by Artisan Components [2]. The table provides a breakdown of the area and timing for each stage, displaying the size of the TPM and CM memories, the ModuleController, and pipeline register, as well as the percentage of the total design corresponding to the ModuleController. We see that the size of each stage within the design is roughly twice that of the previous stage, as expected since the sizes of both the TPM and CM memories are doubled. Furthermore, for a ProMem with 255 entries, the ModuleController only consists of 3% percent of the total design size. Furthermore, the larger the size of the ProMem, the more efficient it becomes, as the overhead of the ModuleController becomes much smaller compared to the size of the memories.

Table 1 presents the access time for each stage of our 8-stage ProMem design. For all stages, an access time of 4 ns was achieved. More importantly, as each stage within the ProMem grew in size, the access time did not increase, in contrast to a CAM design. Furthermore, the access time of ProMem is currently only limited by the access time of the memories for the largest stage within the design. Hence, we can achieve a faster ProMem design by using faster memories within each stage.

With respect to power, notice that our implementation involves only one memory lookup and one comparison per stage. Thus, ProMem does not suffer from the high power consumption that a large fully associative CAM would require do to simultaneously comparing every word in the memory with a key.

5. Conclusions and Future Work

Profiling is a key to numerous design problems that optimize a program and/or architecture. An SOC platform that includes on-chip profiling hardware, along with a software interface, can enable profiling in real-time embedded environments. We introduced a new memory architecture based on a pipelined binary search tree that can monitor a bus for patterns on every clock cycle. The memory has a simple interface to the monitored bus, and scales very well as the memory gets larger. Such profiling could be useful in a variety of scientific applications.

6. Acknowledgements

This work was supported in part by the National Science Foundation (CCR-9876006), the UC MICRO program, and a Department of Education GAANN fellowship.

7. References

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

[2] Artisan Components, Inc. UMC .18 Technology Library, http://www.artisan.com, 2002.

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

[4] Graham, S.L., P.B. Kessler and M.K. McKusick. gprof: a Call Graph Execution Profiler. SIGPLAN Symposium on Compiler Construction, pp. 120-126, 1982.

[5] Lysecky, R., S. Cotterell, F. Vahid. A Fast On-Chip Profiler Memory. Design Automation Conference (DAC), pp. 28-33, June 2002.