A Fast On-Chip Profiler Memory

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

ABSTRACT
Profiling an application executing on a microprocessor is part of the solution to numerous software and hardware optimization and design automation problems. Most current profiling techniques suffer from runtime overhead, inaccuracy, or slowness, and the traditional non-intrusive method of using a logic analyzer doesn't work for today's system-on-a-chip having embedded cores. We introduce a novel on-chip memory architecture that overcomes these limitations. The architecture, which we call ProMem, is based on a pipelined binary tree structure. It achieves single-cycle throughput, so it can keep up with today's fastest pipelined processors. It can also be laid out efficiently and scales very well, becoming more efficient the larger it gets. The memory can be used in a wide-variety of common profiling situations, such as instruction profiling, value profiling, and network traffic profiling, which in turn can be used to guide numerous design automation tasks.

Categories and Subject Descriptors
B.3.0 [**Memory Structures**]: General.

General Terms
Performance, Design.

Keywords
Profiling, system-on-a-chip, platform tuning, adaptive architectures, low power, embedded CAD, binary tree, memory design, embedded systems.

1. INTRODUCTION
Profiling an application executing on a microprocessor is a technique needed to solve a wide variety of optimization problems. 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 has also been used to refer to determining the relative frequencies of values that a variable takes on during program execution.

Profiling appears as part of the solution for a tremendous variety of program and hardware optimization and design automation problems. For example, profiling has long been used to find the most frequently executed subroutines of an application, so that a programmer might focus on optimizing those subroutines [10]. Profiling has been used in compilers to map frequently executed code and data to non-interfering cache regions [15] to improve performance. The approach in [8] proposes using profiling to generate alternate subroutine versions for common cases, with the program then using run-time profiling to pick the best version. Likewise, the approach in [14] uses profiling information to synthesize hardware optimized to the most common situations. Dynamic binary translation methods profile in order to store the translation results of frequent code regions, for improved performance as well as power [13], while dynamic optimization methods search for the hottest blocks for runtime recompilation [3]. The approaches in [4][9] use profiling to detect frequent loops to map to a special address region that an architecture would then map to a small low-power loop cache, while the approach in [12] compresses those regions to reduce memory traffic and hence power. The approach in [6] profiles values of variables or subroutine parameters to detect pseudo-constants that can aid a compiler in optimizing for performance, or even for reduced energy [7].

Most previous profiling approaches, being intended for desktop computing systems, introduce runtime overhead. In particular, either they insert additional code into the application binary, or they interrupt the processor at particular intervals to sample the processor's registers. However, for embedded systems, runtime overhead is often not acceptable, since very tight real-time constraints must be met. Thus, embedded system designers in the past relied on logic analyzers to non-intrusively profile an executing application – though even this was cumbersome and hence not a common feature in design automation techniques.

The trend of increasing chip transistor capacity has led to systems-on-a-chip (SOCs). While providing tremendous advantages in terms of cost, size, performance and power, SOCs have the drawback of low accessibility to the internal components. Thus, logic analyzer probes cannot be connected to arbitrary buses inside the SOC to achieve profiling. Although SOCs typically come with means for accessing internal registers through external pins (e.g., using the JTAG standard [11]), such access is accomplished by stopping normal application execution and then serially scanning the register contents in or out. This access approach incurs large runtime overhead, being intended for test and debug purposes rather than profiling.

Fortunately, the same transistor capacity trend that has led to SOCs has enabled hardware-based approaches to profiling. While on-chip profiling hardware in the past has been limited to high-volume high-performance microprocessors, such hardware can today be added to embedded system prototyping platforms. *Platforms* [16][18] are predesigned SOCs targeted to particular application domains, like set-top boxes, network switches, digital cameras, etc. While some platforms are oriented towards implementation in actual products, others are intended specifically for prototyping. These prototype-oriented platforms are intentionally designed larger than necessary, to accommodate the widest possible range of applications. Thus, adding a relatively

DAC 2002, June 10-14, 2002, New Orleans, Louisiana, USA.

small amount of hardware would likely not be an issue, especially since such platforms are specifically designed for use during the design stage, when profiling would be most needed.

Designing the additional hardware that profiles the executing application non-invasively yet accurately is non-trivial, since updating profile counts fast enough is hard. Existing fast memory designs, even advanced content-addressable memories, have several limitations (which we will describe). Thus, prior profiling hardware typically only counted easy to detect events (like a cache miss) or could be configured to detect only one particular pattern on a bus (like a specific address).

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 tree architecture, our profiler memory achieves single-cycle throughput, meaning complete accuracy even when monitoring very fast sequences of patterns, such as the address bus of modern processors that fetch one instruction per cycle. The binary tree is implemented with a separate module for each tree level, and thus scales well, becoming even more efficient the larger it gets.

In this paper, we define the on-chip profiling problem and discuss related work. We introduce our profiler memory architecture, and describe its use. We show how the memory's modular structure results in timing and area efficient layouts for any size memory. We provide data on the memory's size and performance, and illustrate its use on several benchmarks.

2. PROBLEM DEFINITION

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, meaning we are monitoring the bus before any translation to cache addresses (if a cache exists). 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 translated further to virtual addresses. We assume target addresses could appear 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 that the profiling circuit has the luxury of running at a higher clock frequency.

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

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

This more general problem formulation can also represent problems other than code profiling. For example, the bus B might consist of both an address bus and data bus, so that we might perform value profiling of a set of variables. The bus could correspond to an internal bus of a router chip, for which we might want to profile the traffic to determine the most common destinations, to optimize their routing table lookup speed [20].

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.

3. PROFILING APPROACHES

We can view the general solution to the profiling problem as involving three parts: detecting patterns, checking if detected patterns match a target pattern, and updating the target pattern counts when a match is found. We categorize solution approaches into two categories: software-based and hardware-based.

3.1 Software-Based Profiling

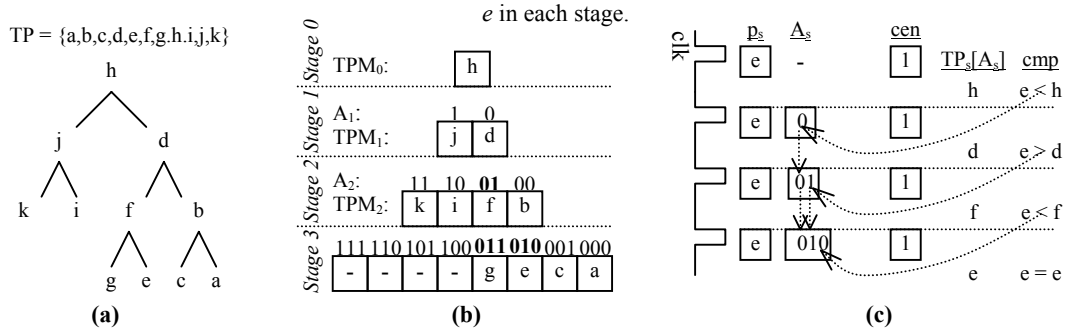
Software-based profiling approaches use the instruction-execution resources of the processor itself to solve the profiling problem.

A common software-based approach involves "instrumenting" the application by adding code to count frequencies of the desired code regions [10]. For example, if we wish to count the frequency of execution of a subroutine, we can add code to the beginning of the subroutine that increments a counter variable declared for that subroutine. This approach is straightforward and flexible, yet incurs significant runtime overhead, especially if the granularity of the code regions being profiled is fine. There are additional problems with this approach in an embedded system. For example, the additional code increases code size, which may not be allowable if the non-instrumented program already used all the program memory (a common situation). The additional code may also cause register spills, thus further modifying program behavior. Furthermore, instrumenting requires a special compiler or a post-compilation binary instrumentation tool.

To reduce runtime overhead, other profiling approaches use sampling techniques [1][8]. Such methods interrupt the microprocessor at certain intervals, and then read the program counter and other internal registers. These methods tradeoff accuracy for reduced overhead. However, in embedded systems, even the reduced overhead may not be acceptable. Furthermore, while the overhead may be reduced, the disruption of runtime behavior during the interrupt itself can be very significant in a real-time system. A related method assumes the existence of a multi-tasking operating system. These methods add an additional task to perform profiling, in place of an interrupt [22]. However, the disadvantages are the same as the interrupt approach.

Others use a simulation-based approach to profiling [5]. The application is run on an instruction-set simulator, with the simulator keeping track of profile information. While accurate, this approach is extremely slow, especially when simulating a system-on-a-chip. Simulating an application for several hours may cover only a few seconds of real time, so this may not exercise realistic code execution. Furthermore, setting up such simulations can be difficult if not impossible for embedded systems, due to the complex external environments that must also be modeled.

Figure 1: Conceptual view of ProMem: (a) sorted binary tree for target pattern set, (b) storing each level (stage) in target pattern memories (TPM) with corresponding memory addresses *A* shown, and (c) example showing how address is formed searching for pattern *e* in each stage.



3.2 Hardware-Based Profiling

Due to the above limitations, embedded system designers have often resorted to using a logic analyzer, where the analyzer's probes are placed directly on the bus to be monitored. The logic analyzer can store a pattern trace, which can be later processed to derive profile information. To avoid the problem of enormous traces filling up available trace memory, certain patterns can be programmed into the analyzer, which the analyzer uses to trigger trace storage for a given period. To perform profiling as we defined earlier, one could simply input the target patterns as triggering patterns. However, a drawback of logic analyzers is their high cost and their ineffectiveness in the era of SOCs.

Some microprocessors include on-chip hardware to assist software developers to profile an executing program [19][21]. Such hardware consists primarily of event counters. Monitored events typically include cache misses, pipeline stalls, branch mis-predictions, and so on. Some counters may be configured to detect a particular bus address, but there are typically only a few of these since having many of them would be costly in terms of size. Thus, the programmer must dynamically reconfigure those counters during program execution to obtain a more complete profile. Not only can this lead to inaccuracy, but also reconfiguring those counters requires additional software instructions, thus bringing us back to some of the problems with software-based profiling.

Recently, a programmable coprocessor for profiling has been proposed, primarily intended to detect performance-related events in high-performance processors [23]. The memory we propose could be used in conjunction with such a coprocessor to be able to detect large pattern sets on a bus.

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. There are several CAM implementation types. One type uses a fully associative memory, which simultaneously compares every location with the key. Fully associative memories do not scale well to larger memories – they become very large and difficult to place and route due to all the comparison logic, and their access time increases due to the high capacitive load involved in sending the key to all memory addresses simultaneously. Thus, newer versions of CAMs use a regular static or dynamic RAM coupled with a smart controller. The controller implements an efficient lookup data structure in the memory, such as a binary tree, or a Patricia Trie [20]. Lookups of binary trees are of the order of log of the number of data items, while lookups for a Patricia Trie are linear with respect to the number of characters (or bits) in the key.

CAMs thus do not satisfy our goals for a profiler memory. While CAMs built using fully associative memories may have single-cycle access time, they do not scale well to larger target pattern sets. On the other hand, CAMs built from regular RAM require multiple cycles for lookup, which does not support the situation where target patterns appear on the bus on every cycle.

Likewise, hash tables do not satisfy the goal either, since they require multiple cycles to compute the hashing function, and may require additional cycles in case of a collision. Finally, approaches that queue the input patterns only help if such input does not appear on every cycle.

4. 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. The memory need not actually update the count field of a matching target pattern until many cycles later.

4.1 Pipelined Binary Tree

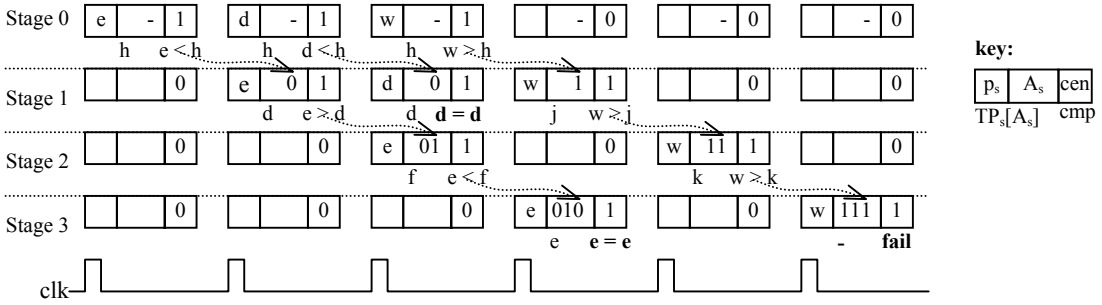
To achieve a throughput of one new pattern every cycle, we implement our ProMem memory structure using a pipelined binary tree structure to find the location of the given input pattern. Figure 1(a) shows an inorder binary tree for a target pattern set *TP*={a, b, c, d, e, f, g, h, i, j, k}. We will implement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

Before presenting the architecture of each module, we will describe conceptually the manner in which each module searches for matches. Figure 1(c) illustrates the search for a single input pattern *e* in the tree of Figure 1(b). We compare *e* with the root node *h*. Since $e < h$, the address A_1 passed to stage 1 is 0. At stage 1, we compare *e* with $TPM_1[0]$, or *d*. Since $e > d$, the address A_2 passed to stage 2 is 01. At stage 2, we compare *e* with $TPM_2[01]$, or *f*. Since $e < f$, the address A_3 passed to stage 3 is 010. Finally, at stage 3, we compare *e* with $TPM_3[010]$, or *e*. Since equal, we found a match, and we then increment a count value associated

Figure 2: Example search for input patterns P = {e, d, w}, showing pipelined behavior of ProMem.



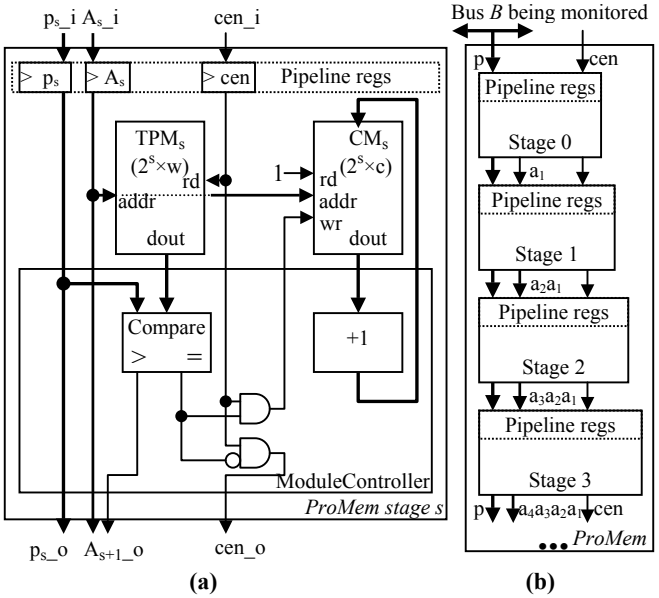
with *TPM₃[010]*. Those count values are actually stored in a separate memory in a stage's module, as we will see later.

The above example illustrated how one item would be searched for in the tree. Figure 2 extends the example by continuing to feed input patterns to the root node on each clock cycle, illustrating the pipelined and hence single-cycle throughput of ProMem. We wish to search for input patterns *e*, *d*, and *w*, which appear in that order on a bus during consecutive clock cycles. We see the same sequence as the above example for *e* as it progresses through the pipeline, though in this figure we can also see how the enable signal *cen* is updated. On the second clock cycle, we see *d* enter the pipeline at Stage 0, from where it is passed to Stage 1 as the right child. In Stage 1, a match is found. Notice that this match disables the continued search during subsequent cycles in Stages 2 and 3, by setting *cen* to 0. *w* progresses through the tree following left children, failing when it hits a node with no target pattern stored.

4.2 Memory Architecture

Figure 3(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.

Figure 3: 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).



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. Such a memory will be slightly larger than a single port memory but not as large as a true dual port memory. 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, indicating we have not found the target pattern and need to continue searching.

To address 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, it is concatenated with single bit with 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 it 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 the binary tree as no valid patterns will be found within that sub-tree.

Although the above design can be used for each stage of our binary search tree structure, using this design for the first stage would not be efficient. With only a single entry within both the TPM and CM, implementing the TPM and CM as actual memories would be inefficient. Instead, the TPM and CM of the first stage are implemented using two registers. In addition, the input to the first stage will only consist of the enable signal, *cen_i*, and the target pattern. All other computation and logic will be implemented in the same manner as the remaining stages.

To construct the entire ProMem structure, we simply connect a module to the module for the next stage, as shown in Figure 3(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.

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**

5. ADDITIONAL CONSIDERATIONS

We thus far have introduced the main feature of ProMem, namely its pipelined structure enabling single-cycle throughput when monitoring a bus. We now discuss how to write the target patterns into ProMem, how to read out the counts after the monitoring period is complete, extensions to support ranges, and layout considerations.

We initially considered memory mapping every internal register. The memory driver software could then store the target patterns, and read the counts, in any manner desired. This option has the undesirable requirement that all the modules be connected to a single bus, whether a processor's peripheral bus, or an internal bus connected through a bridge to that peripheral bus. In either case, running a bus to all modules is costly in terms of area. Furthermore, each module would then require additional logic to interface to that bus to detect and process write and read requests.

A second option was to connect all registers in a scan chain. Connecting the registers in sorted order would require excessive wires between stages. Connecting in breadth first order eliminates that problem. Another potential issue is the compatibility of this scan chain with that used for testing, and the potential complexity of the driver software requiring access to JTAG ports. We leave this potentially viable option for future work.

We currently implement a scheme that makes use of the existing pipeline structure, by adding a few additional control lines and an up-counting register *ptr* added to each module. *ptr* is one bit wider than the module's input address. We initialize *ptr* to zero upon memory reset. We add a write enable signal *wen_i* to each module, and a 1-bit register *wen* as part of the module's pipeline register. If a module at stage *s* detects *wen*=1, it writes the input pattern *p* to *TPM[ptr]* and increments *ptr*, but only if the high-order bit of *ptr* was 0. If the high-order bit was 1, this means all the words in TPM have already been written. In this case, the module instead passes the pattern *p* and the *wen* signals to the next stage. Such functionality results in the entire ProMem target pattern contents being written in breadth first order (right to left for each level). Thus, the driver software must provide the target patterns in breadth-first order – a straightforward task. We assume the target patterns can be written to the bus being monitored. Otherwise, a second input for a target pattern would be required *on Stage 0 only*.

In addition to the *wen_i* signal, we also add the *val_i* signal to the pipeline register. This signal will be used to indicate that the target pattern being written is valid. To do so, the *val_i* input and the target pattern will be concatenated together and written to the TPM. It is important to note that the driver software must initialize all locations when loading a new target pattern set to ensure all entries within the TPM contain the correct valid bit.

Reads of the counts are done similarly. However, we must first reset all the *ptr* counters to 0 – we include an additional input signal *rstptr_i* and 1-bit register *rstptr* to each module, which asserts the clear signal on *ptr* and also gets passed on to the next stage. We must therefore hold this signal high and then low for a certain number of cycles (two times the number of stages) to clear all the *ptr* counters. We also add a read enable signal *ren_i* and a 1-bit register *ren*. Reading then results in a breadth first outputting of the target memory contents onto the pattern output of the last stage. Thus, the driver software must reorder this output into sorted order – again, a straightforward task.

ProMem can be extended to support ranges. For example, we may want to count how many times patterns appear with a value between 1 and 10, 11 and 20, 21 and 30, etc. Such functionality could be useful to count, for example, the number of accesses to a large array, or the number of instructions executed within a subroutine or loop. We simply use two TPMs to store the target pattern ranges. The first TPM will contain the lower bounds of the ranges, and the second TPM will contain the upper bounds of the ranges. As before, the target ranges will be stored in sorted order within the binary tree structure (the ranges must be non-overlapping). When searching to detect if an input pattern is part of the target range, we will use two comparators to simultaneously compare the input pattern with both the lower and upper bound of the target range. If an input pattern is not within the target range, the output from greater than comparison with the lower bound will determine the low order address bit for the next stage.

Finally, another important concern in the design of ProMem is the ability to create an efficient layout. We would like to create a layout that is as tight as possible. In addition, we must ensure that the number of wires and length of wires connecting each module is at a minimum. Assuming each module has a rectangular layout, we can tightly place each module of the ProMem in a spiral fashion, with the output put of each stage abutted with the input of the nest stage. However, in addition, we must ensure that the module corresponding to level 0 is placed on the boundary of the entire structure. Furthermore, the register files or memories within each module can come from a standard register file or memory implementation, which already have efficient layouts.

6. RESULTS

We implemented ProMem using 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, it was implemented as its own entity. The remaining stages, however, use a single entity that contains a generic *STAGE*, which specifies the stage of the instantiated entity. Furthermore, all stages contain a generic *WIDTH*, which specifies the *WIDTH* of the target patterns and the associated count values. Using this implementation approach, 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. The ProMem was synthesized with Synopsys Design Compiler [17] using the UMC 0.18 technology and memory libraries provided by Artisan Components [1]. The table provides a breakdown of the area and timing for each stage, displaying the size of the TPM and CM

Table 2: Comparison of ProMem with fully-associative CAM.

Entries	ProMem (Gates)	CAM (Gates)	% Reduction
1	1682	1006	-67%
3	4288	4081	-5%
7	8481	8319	-2%
15	16058	16627	3%
31	30113	33276	9%
63	57230	68066	16%
127	111802	145100	23%
255	228891	309831	26%

memories, the ModuleController, and pipeline register as well as the maximum clock frequency at which the ProMem can operate. 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.

Another important aspect of our ProMem design is the access time for reading and writing to/from the memories in each stage. 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. The timing results were obtained with Design Compiler using the synthesized design along with the timing information in the UMC technology library.

In addition to the total size of our design, we would like to see how much overhead the supporting logic with our ProMem adds to the overall design. Thus, Table 1 also presents the percentage of the total design corresponding to the ModuleController. We see that the overhead per stage decreases for deeper stages. We also see that for a ProMem with 255 entries, the ModuleController only consists of 3% percent of the total design size.

Table 2 compares our ProMem design to a fully-associative CAM for various numbers of entries. For a small number of entries, the overhead of the ModuleController within the ProMem structure results in a larger size than a design using a CAM. However, for as little as 15 entries, our ProMem design results in smaller overall size than the CAM approach. Furthermore, for a design with 255 entries our ProMem design is 26% smaller than the alternative design. Furthermore, as mentioned earlier, fully-associative CAMs will have slower access time as the size grows since each pattern must be sent to every entry.

7. CONCLUSIONS

Profiling is a key to numerous design automation problems that optimize a program and/or architecture. A prototype-oriented SOC platform that includes on-chip profiling hardware, along with a software interface to that hardware, can enable profiling in real-time embedded environments. We introduced a new memory architecture, based on a binary 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 it gets bigger.

8. ACKNOWNLEDGEMENTS

This work was supported in part by the National Science Foundation (CCR-9876006) and the UC MICRO program.

9. REFERENCES

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

[2] Artisan Components, Inc. UMC .18 Technology Library, http://www.artisan.com, 2001.

[3] Bala, V., E. Duesterwald, and S. Banerjia. Dynamo: A Transparent Dynamic Optimization System. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), June 2000.

[4] Bellas, N., et al. Energy and Performance Improvements in Microprocessor Design Using a Loop Cache. ICCD, pp. 378-383, 1999.

[5] Burger, D. and T. M. Austin. The SimpleScalar tool set, version 2.0. Tech. Rep. CS-1342, University of Wisconsin-Madison, June 1997.

[6] Calder, B., P. Feller and A. Eustace. Value Profiling. MICRO, pp. 259-269, 1997.

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

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

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

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

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

[12] Ishihara, T., H. Yasuura. A Power Reduction Technique with Object Code Merging for Application Specific Embedded Processors. DATE, March 2000.

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

[14] Lakshminarayana, G., et al. Common-Case Computation: A High-Level Technique for Power and Performance Optimization. DAC, pp. 1-5, 1999.

[15] Pettis, K. and R.C. Hansen. Profile Guided Code Positioning. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), June 1990.

[16] Semiconductor Industry Association. International Technology Roadmap for Semiconductors: 1999 edition. Austin, TX: International SEMATECH, 1999.

[17] Synopsys, Inc. Design Compiler, http://www.synopsys.com, 2001.

[18] Vahid, F., T. Givargis. Platform Tuning for Embedded Systems Design. IEEE Computer, Vol 34, No. 3, pp. 112-114, March 2001.

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

[20] Waldvogel, M., et al. Scalable High Speed IP Routing Lookups, SIGCOMM 97, 1997.

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

[22] Zhang, X., et al. System Support for automatic Profiling and Optimization. Proceedings of the 16th Symp. on Operating Systems Principles, 1997.

[23] Zilles, C.B. and G.S. Sohi. A Programmable Co-processor for Profiling. International Symp. on High-Performance Computer Architectures, 2001.