Accelerating Search and Recognition with a TCAM Functional Unit

Atif Hashmi and Mikko Lipasti
Department of Electrical and Computer Engineering, University of Wisconsin - Madison
ahashmi@wisc.edu, mikko@engr.wisc.edu

***Abstract*— World data is increasing rapidly, *doubling* almost every three years[1][2]. To comprehend and use this data effectively, search and recognition (SR) applications will demand more computational power in the future. The inherent speedups that these applications get due to frequency scaling will no longer exist as processor vendors move away from frequency scaling and towards multi-core architectures. Thus, modifications to both the structure of SR applications and current processor architectures are required to meet the computational needs of these workloads.**

This paper describes a novel hardware acceleration scheme to improve the performance of SR applications. The hardware accelerator relies on Ternary Content-Addressable Memory and some straightforward ISA extensions to deliver a promising speedup of 3.0-4.0 for SR workloads like *Template Matching, BLAST*, and multi-threaded applications using *Software Transactional Memory (STM)*.

I. Introduction

RECENT studies at the University of California-Berkeley[1][2] show that world data is *doubling* every three years and is presently measured in exabytes-a billion billion bytes. To make meaningful use of this enormous amount of data, SR applications are gaining in popularity[1]. As processor vendors move away from frequency scaling and towards multi-core architectures, the inherent speedups that SR workloads get with an increase in processor speed will diminish in the future. Therefore, to meet the computational demands of SR workloads, substantial modifications to both the structure of SR applications and architecture of current processors are required.

Table I shows the percentage of time spent in performing the search operation by some SR applications. These results suggest that improving the performance of search operations can significantly speedup these applications.

TABLE I
PERCENTAGE OF TIME VARIOUS APPLICATIONS SPEND IN PERFORMING THE SEARCH OPERATIONS

Application	**%age Time Spent in Search**
Template Matching	95%
BLAST	65%
Software Transactional Memory	60%-95%

This paper describes a novel hardware acceleration scheme to improve the performance of SR applications by accelerating the search operation. The proposed Ternary Content-Addressable Memory (TCAM) Functional Unit (TFU) makes use of banks of TCAM and is operated using straightforward ISA extensions. Our proposed design for the TFU and the ISA is not targeted to any single application. Rather, the TFU and the ISA provide primitives and not the exact solution, enabling a broad array of applications from the SR domain to efficiently utilize the proposed TFU. Simulation results show that the TCAM-based functional unit and the proposed ISA delivers a promising speedup of 3.0-4.0 for a set of representative benchmark applications.

The main contributions of our work are as follows.

- We designed a novel, inexpensive, and efficient TCAM-based functional unit.
- We propose ISA modifications that take advantage of our novel TCAM architecture.
- We provide area and power estimates for the TFU.
- We demonstrate the general-purpose nature of the ISA extensions by utilizing it in a variety of ways.
- We implement and evaluate a representative suite of benchmarks from several different domains that obtain substantial speedups.

II. Design of the TCAM Functional Unit

This section describes the overall architecture of the TFU and discusses various factors that motivated the final design.

A. Basic TCAM Architecture

TCAMs are data addressable memories that search a query among all the entries stored in the TCAM memory. The query contents are searched in parallel with all the entries in the TCAM and the output of this match operation is fed to the priority encoder which provides the location of the first match. Bits in the TCAM can be individually masked. The masked bits are then treated as *Don't Cares* during search. Figure 1 shows the block level diagram of a TCAM.

B. Structure of SR Applications

There are three main components of any TCAM-based SR application.

- Split a large query into multiple overlapping or non-overlapping chunks.
- Exhaustively search each of the chunks within all the entries of the database.
- Based on a predefined metric, determine where the best match occurred.

To speedup the SR applications, the TFU should have a TCAM bank to store the database entries. Since the

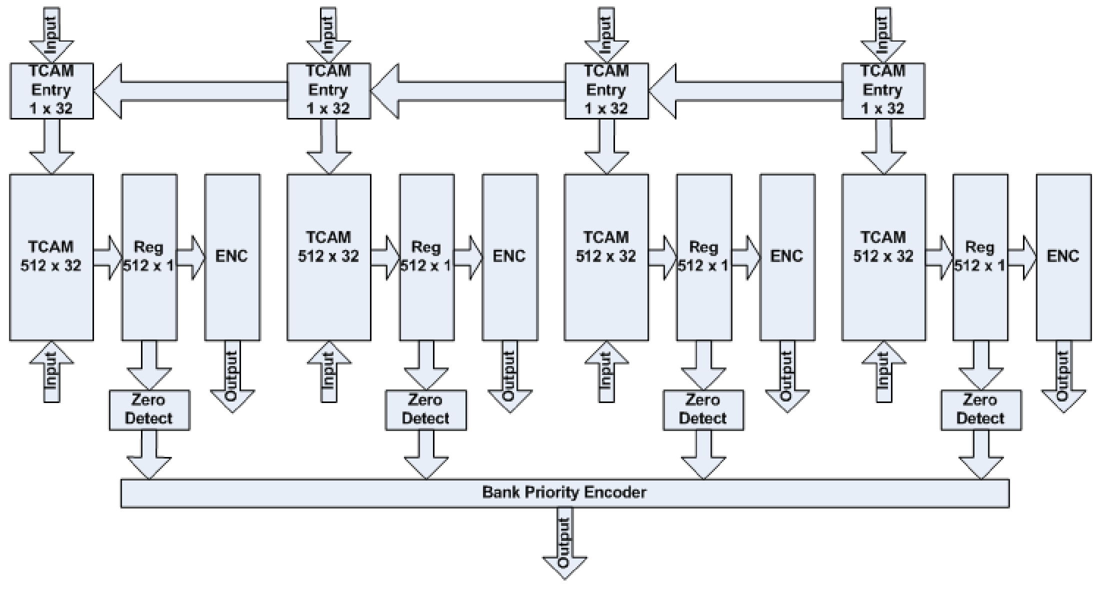

Fig. 3. Block level diagram of the proposed TCAM functional unit. *Position Register* is merged into the TCAM for the sake of image clarity.

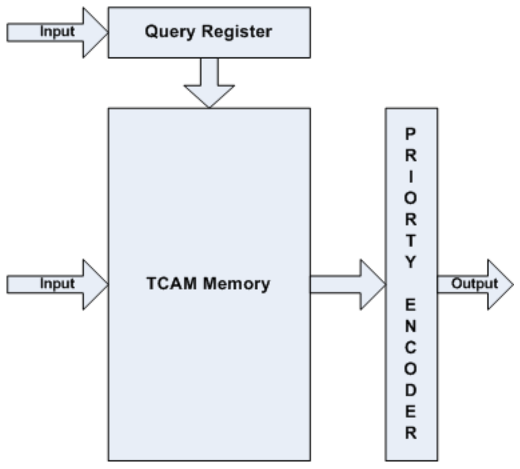

Fig. 1. Block level diagram of a single TCAM bank.

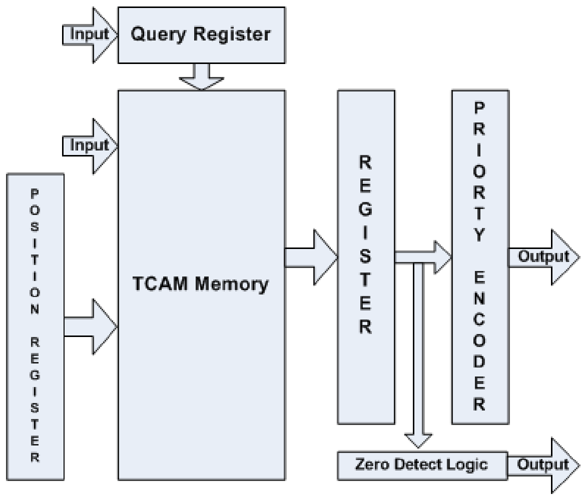

Fig. 2. Block level diagram of modified TCAM bank.

TCAM architecture described in Section II-A can notify an application about the location of only the first match and not about subsequent matches, it is not suitable for most of the SR applications. To solve this issue, we need to have a register between the TCAM memory and the priority encoder so that all the match/mismatch information can be registered. We call it a *match register*.

To increase the throughput of the TFU, multiple TCAM banks should be available. This also enables parallel searches of the same query at multiple locations within the same TCAM memory. There is one issue involved with having multiple banks: we need a path between adjacent query registers to shift the query from one register to the other to get a match at bank boundaries.

To further improve performance, the output of the match register is also fed into a zero-detect circuit which tells us whether the register has any non-zero bits.

C. Modified TCAM Architecture

Discussion in Section II-B shows that slight modifications to the architecture of a traditional TCAM bank are required. These modifications include a Kx1 bit register in between the TCAM and the priority encoder. Here, K is the number of rows in the TCAM. The query register can also store *Don't Cares*. This is done in order to increase the flexibility of the system because it removes any limits from the minimum size of the query. There is a *position register* attached to the TCAM which is simply a counter and it keeps track of next available entry in the TCAM. Figure 2 shows a modified TCAM bank.

D. Architecture of TCAM Functional Unit

A block level diagram of the proposed TCAM functional unit is shown in Figure 3. Multiple modified TCAM banks are used so that a query can be searched in parallel in all the TCAMs. This increases the overall throughput of the system because the query is searched in different parts of the database at the same time. Our initial sensitivity studies led us to a cost-effective design with 4 TCAM banks.

Initially, all the TCAMs are loaded with the database entries and the query registers are loaded with the query.

The query slides over the TCAM banks with the database entries in it and at each step, match/mismatch information is stored into the output register. The priority encoder attached to each register gives the location of the first match. The output of the zero-detect logic is fed into the *Bank Priority Encoder*, which tells us the first TCAM bank which has a match.

III. TCAM Overheads

To approximate the total overhead added to the processor by the TFU, we estimated its area cost and power consumption.

To calculate the approximate area of our proposed accelerator, we obtained the area of TCAM cell implemented in 130 nm CMOS technology from Noda et al.[4]. The areas of the output register, priority encoder, zero-detect logic, and position were calculated by implementing the basic blocks used by them in *Cadence* design tool using 400nm CMOS technology. Using the areas of these basic blocks, the areas of all the components of the TCAM accelerator were estimated. Finally, the area of the entire TCAM accelerator was estimated by adding the areas of the individual components. All the areas were then optically scaled down to 90nm technology so that the total area of the TCAM accelerator could be compared with the area of an L1 cache implemented in 90nm technology.

Table II shows the area calculated for various components of the TCAM accelerator for 90 nm technology. The areas were scaled by multiplying the area of unit component with the number of instances of that component. For example, Noda et al.[4] showed that the area of a single TCAM cell is 4.79 um^2. This means that for a 512x32 bit TCAM, the area would be $512 \times 32 \times 4.79 \approx 78479\, um^2$. A common practice among the VLSI community is to add a 30% routing overhead when ever the area is scaled from a unit component to multiple instances of the same component. Adding this routing overhead to the total area of the TCAM, we get $78479 \times 1.3 = 102023\, um^2$ which scales down to $48898\, um^2$ for 90 nm technology. The areas were scaled down by using the linear scaling model described by Borkar[5].

TABLE II
Area calculation for TCAM accelerator using 130nm and 400nm technologies.

Component	**Area at 90nm** (um^2)
Query Reg.(32 bits)	95
Position Reg.(9 bits)	276
Output Reg.(512 bits)	13996
Priority Enc.(512 to 9)	22006
Zero-Detect(512 to 1)	67
Bank Enc.(4 to 2)	95
TCAM (512x32 bits)	48898

Using the areas of various components shown in Table II, we can estimate the area of the TCAM accelerator in the following way.

$$\begin{aligned} &Total\,Area\,of\,TCAM\,Accelerator = \\ &\quad 4 \times Area\,Query\,Reg + 4 \times Area\,Output\,Reg + \\ &\quad 4 \times Area\,Priority\,Enc + 4 \times Area\,Position\,Reg + \\ &\quad 4 \times Area\,Zero\,Detect + Area\,Bank\,Enc + \\ &\quad 4 \times Area\,TCAM \end{aligned}$$

$$Total\,Area\,of\,TCAM\,Accelerator = 341447\, um^2 = 0.341\, mm^2$$

Adding the 30% routing overhead, we get:

$$Total\,Area\,of\,TCAM\,Accelerator = 0.443\, mm^2$$

A typical processor might have separate 32KB L1 data and instruction caches. The size of these caches are generally around 1.0-2.0 mm^2[6]. This indicates that the area of our proposed accelerator is significantly smaller than a current generation L1 cache. Thus, implementing the accelerator on the same die as the processor will result in reasonable area overhead.

To calculate the power consumed by the TCAM accelerator, we used the TCAM power estimation model proposed by Agrawal et al[7]. The TCAM bank architecture used by Agrawal et al. for power calculations is slightly different than the one we have proposed. First, their architecture does not include a register between the TCAM and the priority encoder. Second, they do not model the zero detect logic. Their simulation results show that around 98% of a TCAM bank's power is dissipated by the search and match lines in the memory. Thus, their power model can give us a rough estimate of the total power consumed by our TCAM accelerator. We calculated the energy dissipated per access by our accelerator, configured according to Table II. The total energy consumed per access came out to be 0.14 nJ for a 90nm technology with 1.5 V power supply. It should be noted that the power model proposed by Agrawal et al. assumes 100% activity factor and as a result provides the worst case power consumption.

IV. CPU TFU Interaction

The proposed TFU is not part of the main processor pipeline; rather it is accessed in the same manner as the L1 cache. Thus, the TFU does not affect the performance of an application not making use of it. In order to populate the TFU, the processor moves the data from L1 cache to the TFU memory. Once the data is moved to the TFU, it stays there till it is overwritten. Since the TFU is populated once and then various search patterns are applied to it, the TFU can prevent a significant number of conflict and compulsory cache misses. This secondary effect can further improve the performance of the applications making use of the TFU.

V. ISA for the TCAM Functional Unit

Table III details the ISA for the efficient use of the features provided by the TFU. Using this ISA, software developers can make use of the primitives provided by the TFU.

TABLE III
ISA EXTENSIONS FOR OPERATING THE TCAM FUNCTIONAL UNIT.

Instruction	Input	Description
AddEntryToTCAM	Bank#, Value	Writes a value to the specified TCAM bank
SetTCAMEntryMask	Bank#, Mask, Bank Row #	Sets the mask of an entry in the specified TCAM. Mask determines which bits should be treated as *Don't Cares.*
AddEntryToQueryRegister	Bank#, Value	Adds an entry to the query register of the specified TCAM bank.
SetTCAMQueryRegisterMask	Bank#, Mask	Sets the mask of the query register in the specified TCAM bank. Mask determines which bits should be treated as *Don't Cares.*
PerformSearch		Latches the match/mismatch information of a TCAM bank into the output register.
ShiftTCAMQueryRegisters		Shifts the values and the masks of the query registers by one character i.e. 8-bits.
SetTCAMPositionRegister	Bank#, Value	Updates the contents of the *position register*. This allows the programmer to modify TCAM entry of its choice.
ReadPriorityEncoder	Bank#	Returns the value of the priority encoder of the specified TCAM bank.
ClearTCAMFirstOne	Bank#	Clears the very first non-zero bit of the output register of the specified TCAM bank. When the first one is cleared the priority encoder provides the location of the next one. Thus, the application is not required to traverse the entire match register one bit at a time to obtain the match/mismatch information.
ReadTCAMZeroFlag	Bank#	Returns the output of the zero-detect logic of the specified TCAM bank.
ReadTCAMBankEncoder		Returns the output of the *bank priority encoder.*

VI. IMPLEMENTING SR APPLICATIONS USING TCAM ISA

Figure 4 shows the pseudo-code for a sample program that reads integers from a file and provides locations of all integers in the file that are between 512 and 1024. Pseudo-code for implementing the same application using the TCAM ISA is shown in Figure 5. For the TCAM-based implementation, we have used the same configuration as shown in Table II. For the sake of simplicity, we have assumed that the number of entries in the file is a multiple of 2048.

```
$fd = \text{open}(filename);$
$location = 0;$
while(! $endoffile$)
begin
  $valueRead = \text{read}(fd, \text{sizeof(int)});$
  if($valueRead >= 512 \&\& valueRead < 1024$)
    print($location$);
  $location = location + 1;$
end
```

Fig. 4. Pseudo-code for an application that searches an integer value provided by the user within all the integers stored in a file and provides the locations of all the matches.

In the TCAM ISA-based version, initially the query registers are loaded with the value of 512. Then the TCAMs are populated with values from the file and for each location, the mask is set to 0xfffffe00. This means that we are interested in the higher 23 bits only of the value stored in the TCAM. A value between 511 and 1024 will have zeros at bit locations 10-31 and there will be a one at location 9. The lower 9 bits do not matter. The *PerformSearch* instruction latches the match/mismatch information to the output register. The most interesting part of the code are the instructions *d1* and *d2*. In *d1*, we read the value of the priority encoder attached to the output register. This gives us the location of the first match. Next, we clear that bit by using the *ClearTCAMFirstOne* instruction. This gives us the location of the next match. The *inner while loop* will iterate as many time as the number of matches. This behaviour of the TFU significantly improves the

```
for $i = 1$ to $4$
begin //$Loading the Query Registers$
  AddEntryToQueryRegister($i$, 512);
  SetTCAMQueryRegMask($TCAM$,
  $0xffffffff$); //$Setting the don't care bits$
end
$fd = \text{open}(filename);$
while(! $endoffile$)
begin
  for $i = 1$ to $2048$
  begin //$Populating the TCAMs$
    $value = \text{read}(fd, \text{sizeof(int)});$
    $TCAM = \lfloor (i \div 512) \rfloor;$
    $loc = (i\%512);$
    SetTCAMPositionRegister($TCAM, loc$);
    AddEntryToTCAM($TCAM, value$);
    SetTCAMEntryMask($TCAM, loc$,
    $0xfffffe00$); //$0$ $means bit is don't care$
  end
  PerformSearch();
  for $i = 1$ to $4$
    while(ReadTCAMZeroFlag(i))
    begin
      d1: print(ReadPriorityEncoder($i$));
      d2: ClearTCAMFirstOne($i$);
    end
end
```

Fig. 5. Pseudo-code for the TCAM ISA-based implementation of program shown in Figure 4.

performance of the TCAM ISA-based implementation as compared to the software only implementation. The TCAM ISA-based implementation is not limited by the size of the TCAM. No matter what the number of integers is in the file, due to the TCAM ISA the TCAMs can be loaded multiple times until the entire file is searched.

To test the functional correctness of our proposed TFU and to estimate the speedups obtained, we implemented *Template Matching*, *BLAST Stage I*, and *STM* using the TCAM ISA. Due to the page limits, pseudocodes for these applications are not presented in the paper.

TABLE IV

TIME TAKEN BY DIFFERENT TCAM FUNCTIONAL UNIT INSTRUCTIONS.

Instruction	Time (nsec)	CPU Cycles
AddEntryToTCAM	30	75
SetTCAMEntryMask	30	75
AddEntryToQueryRegister	30	75
SetTCAMQueryRegisterMask	30	75
PerformSearch	10	25
ShiftTCAMQueryRegisters	10	25
SetTCAMPositionRegister	5	12
ReadPriorityEncoder	5	12
ClearTCAMFirstOnea	5	12
ReadTCAMZeroFlag	5	12
ReadTCAMBankEncoder	5	12

VII. SIMULATIONS AND RESULTS

This section describes our simulation methodology to evaluate our design and to estimate the possible speedup for applications that use our proposed TFU. For all our simulations, we used a 2.5 GHz Intel P4 processor with 512 MB RAM.

A. TCAM Functional Unit Simulator

To validate the functional correctness of the applications developed using the TCAM ISA and to estimate the speedups obtained, we developed an event-driven simulation model. The simulator implements the entire TFU in details and provides the TCAM ISA as an API. It keeps track of various TCAM events generated by an application that uses the TCAM ISA and also logs the total CPU time spent by the application on the TCAM events. Upon completion, the time spent on the TCAM events is subtracted from the total execution time of the application and an estimate of the time spent by the TFU is added. The estimate of time spent by the TFU is obtained by assigning an amount of time to each of the TCAM operations as shown in Table IV. Since, the unit is on die and is significantly smaller than an L1 cache, the TCAM events can complete in less than 10 CPU cycles. However, as is clear from Table IV, we have assumed that even the simplest TCAM instruction takes more than 12 CPU cycles to complete. Despite these pessimistic assumptions, we obtain very promising results.

B. Template Matching

In Figure 6, we compare the amount of time taken by the software-only implementation of the *Template Matching* algorithm with the TCAM ISA-based implementation. In our experiments, we kept the size of the reference image constant at 2048x2048 and varied the size of the template image. Graph in Figure 6 clearly shows that the TCAM ISA-based implementation is 3-4 times faster than the software-only one.

C. Stage I of BLAST

To evaluate the speedups obtained when the first stage of BLAST is implemented using the TCAM ISA, we compared its execution times with the first stage implemented in WU-BLAST, the fastest implementation of the BLAST so far[8]. For our experiments, we used the

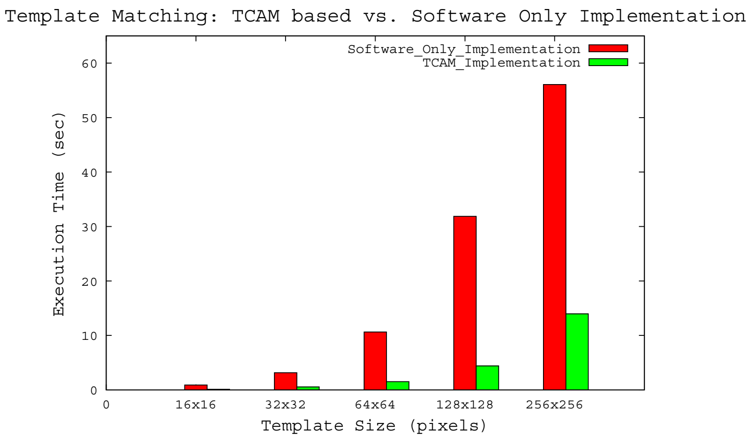

Fig. 6. Comparison between the execution times of software only implementation of *template matching* and the TCAM ISA-based implementation.

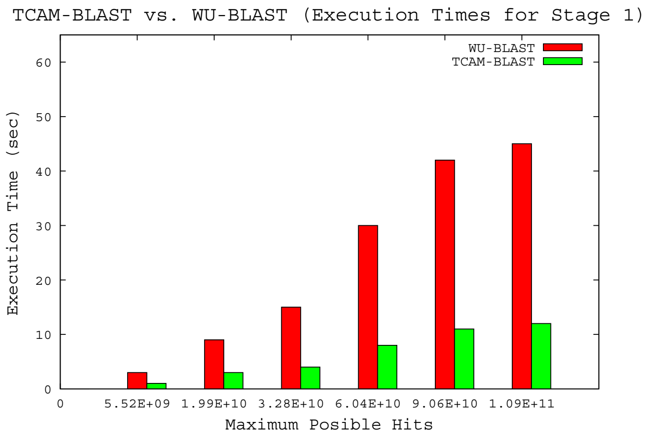

Fig. 7. Comparison between the execution times of first stage of WU-BLAST and its implementation using the TCAM ISA.

est_human database obtained from the NCBI website and queries of various sizes. The size of this database is 4.7 GB. Our simulation results show that TCAM ISA-based implementation is 3-4 times faster than the WU-BLAST implementation of the first stage. The x-axis represents the maximum number of possible hits. This is actually the maximum number of overlapping chunks that can be generated from a query. Figure 7 shows that the TFU can provide efficient searching even in the common case of very large databases.

D. Memory Access Operation of STM

To measure the speedup obtained when the memory access operation of software transactional memory (STM) system makes use of the TFU, we studied two STM-based multi-threaded applications.

The first application calculates the frequency of different numbers stored in a array. The array is equally distributed among all the threads and each thread updates the shared variables keeping track of the frequency of numbers in the array. We call this algorithm the *histogram* algorithm. In our tests, we kept the total size of the data array constant at 1,000,000 elements and we varied the number of unique digits. By increasing the number of unique digits, the number of shared elements between each thread also increases.

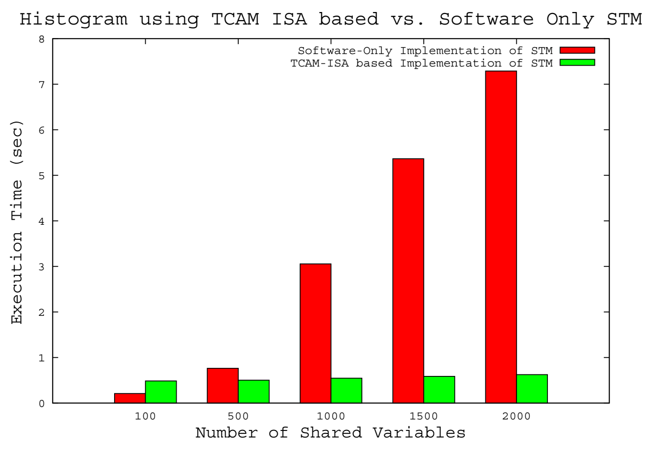

Fig. 8. Comparison between the execution times of the *histogram* algorithm using the software-only implementation and TCAM ISA-based implementation of STM.

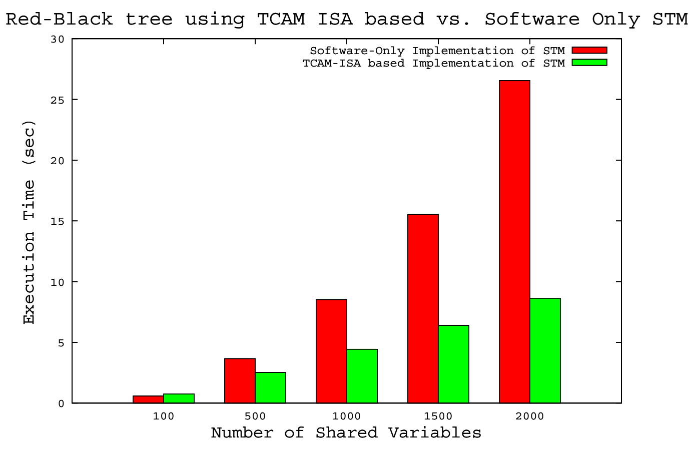

Fig. 9. Comparison between the execution times of the *Red-Black* algorithm using the software-only implementation and TCAM ISA-based implementation of STM.

Total time spent by the *histogram* application using the software-only implementation of STM and the TCAM ISA implementation is shown in Figure 8. Simulation results show that as the number of shared variables is increased, the performance of the version using the software-only implementation of STM degrades significantly and around 10 times speedups are possible if the TCAM ISA is used to implement STM. Since the histogram application spends 95% of its execution time modifying shared variables, the total execution time is directly dependent upon the time spent in the STM function to access shared memory. Since the TCAM ISA-based STM searches an element in constant time, increasing the size of the shared memory has very limited effect on it. When the number of shared variables is 100, the TCAM ISA-based implementation is slower than the software-only implementation due to the overhead of creating a new shared variable; this overhead is higher in the TCAM ISA-based implementation than in the software-only implementation. In the case of software-only implementation, the new shared variable is added to the head of the linked list while in case of TCAM ISA-based implementation, it is added to the next available free slot in the TFU.

The second STM application that we studied was an implementation of *Red-Black* tree. For our experiments, we created trees of various sizes and searched 1,000,000 random numbers in those trees. The graph in Figure 9 shows the average execution time taken to search a number in the *Red-Black* tree. From the graph, we can see that the version using the TCAM ISA-based implementation of STM is 2-3 times faster than the one using the software-only implementation. The speedup is less than the speedup obtained in case of the *histogram* application because in case of *Red-Black* tree, only 60% of the total execution time is spent on accessing the shared variables.

VIII. Conclusions and Future Work

In this paper, we have proposed a general purpose functional unit that utilizes TCAMs. The unit is operated through straightforward ISA extensions and provides significant speedups for search and recognition applications. Our proposed hardware and ISA can be used by a vast variety of SR applications. Software developers can use the ISA to tune the functionality of the unit according to needs of their application. Area and power estimates show that the proposed TFU is 3-4 times smaller than a typical L1 cache and consume 0.25nJ of energy per access in 90nm technology. Thus, implementing the TFU on the same die as the processor will result in very little area overhead and power cost. We verified the functional correctness of our proposed TFU and the ISA by implementing a simulation model for the TFU. The simulation model was also used to conduct the performance studies. We implemented *Template Matching, Stage I of BLAST* and *STM* using the TCAM ISA and compared the performance of TCAM ISA-based implementations with the software-only implementations. Simulation results show that if an application uses the TFU, its execution time can be reduced by a factor of 3-4x as compared to a software-only implementation.

In the future, we are looking forward to extending our work in following directions. First, we are planning to implement more SR applications to study the performance gains achieved if they use the TFU. Second, we also plan to implement the TFU in a cycle accurate simulator in order to conduct the performance studies at a much finer granularity and to study the effect of the TFU on the memory subsystem. Finally, we want to study the overhead of saving the TFU state in case of a context switch.

References

[1] P. Dubey, *A Platform 2015 Workload Model: Recognition, Mining, and Synthesis Moves Computers to the Era of Tera*, Intel Corporation, 2005.

[2] P. Lymen & H. Varian, *How Much Information*, UC-Berkeley Project, 2003.

[3] J. Lewis, *Fast Template Matching*, Vision Interface, 1995.

[4] H. Noda, K. Inoue, H. Mattausch, T. Koide, & K. Arimoto, *A Cost-Efficient Dynamic Ternary CAM in 130nm CMOS Technology with Planar Complementary Capacitors and TSR Architecture*, Symposium on VLSI Cirsuits, 2003.

[5] S. Borkar, *Design Challenges of Technology Scaling*, IEEE International Symposium on Microarchitecture, 2003.

[6] P. ShivaKumar & N. Jouppi, *CACTI 3.0: An Integrated Cache Timing, Power, and Area Model*, WRL Research Report, 2001.

[7] B. Agrawal & T. Sherwood, *Modeling TCAM Power for Next Generation Network Devices*, IEEE International Symposium on Performance Analysis of Systems and Software, 2006.

[8] WU-BLAST, www.blast.wustl.edu/