A Hybrid Markov Model for Accurate Memory Reference Generation

Rahman Hassan, Antony Harris, Nigel Topham, and Aris Efthymiou

***Abstract*—Workload characterisation and generation is becoming an increasingly important area as hardware and application complexities continue to advance. In this paper, we introduce a concise methodology for workload generation for fast and accurate cache design space exploration. The hybrid model we propose uses an adaptation of the Least Recently Used Stack Model to capture key spatio-temporal locality features and a Markov model is implemented to generate an arbitrary length trace with the given workload characteristics through a dynamically ordered FIFO scheduler. Simulation of a variety of application traces from the SPEC2000 benchmark suite demonstrate the cacheability characteristics of the synthetic memory reference stream is generally very well preserved and similar to its original form.**

***Index Terms*– Cache Simulation, LRUSM, Markov Model, Trace Generation.**

I. Introduction

The design space of cache memory is evaluated using execution-driven or trace-driven simulation. Execution-driven simulation can be performed at various levels of abstraction, from the highly abstract algorithmic level to the highly detailed bit-accurate and cycle-accurate RTL [1]. Instruction Set Architecture (ISA) emulators such as ARMulator [2] and SimpleScalar [3] can perform execution-driven cache simulation with a high level of behavioural and timing accuracy. ISA emulation is becoming increasingly fast in generating on-line traces for cache evaluation, but requires architectural models, application source-code, and a development toolkit. An efficient alternative for memory system evaluation is trace-driven simulation. Trace-driven simulators such DineroIII [4] accept a chronological stream of memory references and evaluate miss statistics based on the selected configuration. However, to obtain meaningful results, particularly for multi-level cache designs, large trace files of hundreds of millions of memory references may be required. Furthermore, since each trace is representative of a particular application, the net result is often an enormous trace repository presenting storage and also portability problems. Truncation and/or trace-sampling can be employed to reduce the effective length of a trace, but any truncation procedure will need to account for cache warm-up and may still incur a sizeable storage penalty, while sampling may trade-off accuracy. Trace compression is another option but compression and decompression operations can be lengthy.

Synthetic workload generation can address the problems of maintaining large traces by using a stochastic model to generate an arbitrary number of memory references on-the-fly; the reference stream can then be passed to a trace-driven simulator for system evaluation. Since the reference stream is governed by a probability distribution, an arbitrary length implies fewer references and therefore faster simulation than traditional execution-driven or trace-driven simulation. Unfortunately, existing synthetic reference models generally lack the accuracy necessary for accurate cache design space exploration using concise workload characteristics.

II. Related Work

Denning [6] was one of the earliest to propose the generation of memory references based on their independent probability. From the perspective of cache simulation, a key problem with the proposed Independent Reference Model (IRM) is its failure to capture the locality of memory references inherent in a real trace. Thiebaut [8] looks at the generation of synthetic program traces using an extension of the basic Distance Model [7]. The idea is to model the probability distribution of the jumps between consecutive memory references as a hyperbolic relationship and generate new references using a random walk through an address space. The proposed model requires parameters for the working-set size and locality of reference. The working-set size for a cache is the storage needed at any one time and contains current and recent data. As the working-set changes during the execution of a program, determining a parameter to quantify its size requires a number of simulation runs. Eeckhout [9] proposes an approach of profiling instruction mixes, branches and dependencies to establish a pattern in the memory reference stream. The approach requires detailed trace information to perform the profiling procedures and operand evaluations. The Partial Markov Model (PMM) presented in Agarwal [10] is based on a two-state Markov chain. The first state produces sequential memory references and the second state generates random references. Maintaining a state or switching state is governed by the data captured from the real trace. A problem with the proposed approach is that a single probability threshold for state transitioning does not capture sufficient

Manuscript received January 5, 2007. This work was supported by ARM Ltd and the Engineering and Physical Sciences Research Council (EPSRC) under EPSRC CTA Grant GR/T/18448/01.

R. Hassan is with the Institute for System Level Integration, Alba Centre, Livingston, EH54 7EG, UK. (rhassan@sli-institute.ac.uk).

A. Harris is with ARM Ltd, Sheffield, S1 4EB, UK. (aharris@arm.com)

N. Topham and A. Efthymiou are with the School of Informatics, University of Edinburgh, EH8 9LE, UK. ({npt, aefthymi}@inf.ed.ac.uk).

temporal information. Sorenson [11] highlights the need to capture both spatial and temporal information from a real trace and extends the original work by Grimsrud [14] on three-dimensional plots called locality surfaces for visualising reference locality. Sorenson evaluates some existing synthetic trace models and analyses their performance using the locality surface. Berg [12] captures trace locality by profiling the *reuse distance* and applies the distribution to a probabilistic cache model to estimate the miss ratio of fully-associative caches. The reuse distance is the number of intervening memory references between identical references. The author uses the reuse distance since a memory reference is less likely to remain in the cache the longer it has not been accessed – and it is this likelihood of eviction that the proposed cache model aims to exploit. However, the assumption that the larger the reuse distance, the higher the probability of a cache line eviction is not necessarily true, at least for synthetic trace generation. The intermediate accesses could all be to the same memory location and a large reuse distance would not reflect this pattern. A better measure of locality is presented by Mattson [13] in the form of the Least Recently Used Stack Model (LRUSM). The LRUSM is a natural representation of least recently used behaviour and is based on the *stack distance*, which is the number of unique intervening memory references between identical references and is a very effective measure of temporal locality. Grimsrud [14] analyses the efficiency of the stack distance model in preserving temporal locality using his locality surfaces, while Brehob [15] uses the stack distance model for implementing a probabilistic cache model to evaluate miss ratio. The works of Mattson, Grimsrud, and Brehob do not analyse the efficacy of the LRUSM in the generation of synthetic traces for trace-driven simulation of cache memory, although Sorenson [11] did study the LRUSM and other models and reported poor cache performance. In this paper, we implement an algorithm adapted from the LRUSM and use it to characterise the spatio-temporal characteristics of application workloads. The profile data is passed to a Markov stochastic model which generates memory references through a dynamically ordered FIFO scheduler driven by a pseudo-random number generator.

III. TRACE LOCALITY

An important general rule of program execution is the *90/10* rule [16], [17], which states that 90% of a program's execution time is spent in only 10% of the code. The rule highlights the significance of locality in predicting what instructions and data a program may use and its potential impact on cache performance. Fundamentally, there are two types of locality that a cache exploits to achieve favourable hit rates: temporal locality, which is apparent when identical memory references occur close together in time; and spatial locality, which is present when physically proximate references occur close together in time. A cache can take advantage of temporal locality by keeping recently referenced data as long as possible, while spatial locality is exploited by performing block transfers (line-fills) on a miss.

For a synthetic memory reference model to produce good cache simulation results, it must seek to preserve the original temporal and spatial locality information. We capture spatial locality using block-size granularity and map memory references to cache line numbers. Any unique references mapping to the same cache line are treated as identical references. A probability distribution based on the LRUSM is used to quantify the temporal locality of line-mapped references. Listing 1 describes the pseudo-code of our trace profiling algorithm. We use a dynamically growing integer stack (*S*). For each memory reference *R* we check if it is resident in the stack. If *R* is not found then it is pushed directly to the top of the stack and assigned a stack distance (*sd*) value of *-1*. If *R* is found in the stack then it is removed from its position and then pushed to the top. The depth from which *R* is fetched is the new *sd*. The stack distances are stored in a stack distance string data structure (*SDS*). New line accesses theoretically have a stack distance of ∞ but we use a finite value of *-1* to enable profiling for our trace generation algorithm. We also capture the temporal order of new block accesses (*L*), the size of which we identify as the full working-set size of the application program code. In addition, a count of the total number of references is maintained. A cache line size of 32 bytes is assumed.

SDS is profiled and the resulting probability distribution is integrated to generate the cumulative distribution of stack distance values, *F*, where each element is computed as

$$F_i=P_i+F_{i-1} \quad \text{for } i=[1:\infty] \quad \text{where } F_0=P_0$$

F and corresponding stack distance numbers *SD* are stored as numerically ordered probability vectors.

```
procedure Profile(tracefile)
declareFIFO(SDS)
declareFIFO(L)
declareStack(S)
B:=32
SIZE:=0
LCOUNT:=0
while forever
    R:=nextRef(tracefile)
    if R=null then break
    else
        R:=R/B
        for i:=0 to SIZE
            if R=S[i] then break
        end for
        if i=SIZE then
            sd=-1
            pushBottom(SDS, sd)
            pushBottom(L, R)
            pushTop(S, R)
            SIZE:=SIZE+1
        else
            sd:=i
            pushBottom(SDS, sd)
            temp:=S[sd]
            remove(S, sd)
            pushTop(S, temp)
        end if
        LCOUNT:=LCOUNT+1
    end if
end while
end procedure
```

Listing 1. Trace profiling algorithm.

IV. Trace Generation

The trace generation algorithm is modelled as a Markov chain. A Markov chain is a discrete-time stochastic process that describes the different states a system can assume at successive time intervals. The Markov property stipulates that a state transition depends only on the current state of the system and not on past or future states. We use a two-state Markov chain model with the first state generating new memory references and the second state generating memory references based on a history of previous references. State transitions are governed by the stack distance cumulative probability distribution vector *F*.

Stack distance values are generated using the Inverse Transform Sampling method [18]. A pseudo-random number generator (PRNG) issues a uniformly distributed number in the interval (0:1) that is mapped to a stack distance using its cumulative distribution. The Markov model issues a new memory reference for a stack distance value of *-1* and any other value generates a previous reference. Inter-state and intra-state probabilities are treated as stochastically independent in line with the Markov property. Figure 1 illustrates an overview of the model.

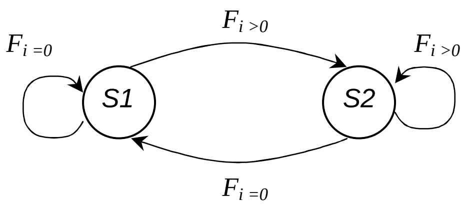

Figure 1. Markov model for state behaviour.

The key to the algorithm is the maintenance of a FIFO data structure that schedules the order of memory references. The FIFO is initialised with the full working-set of memory references mapped as cache line numbers (*L*). On every request for a new reference (state *S1*), the element at the back of the FIFO is popped off and pushed to the front, before being mapped back to a memory reference and passed to the output. On every request for an existing reference (state *S2*), the element at the requested stack distance is read from the front of the FIFO and passed to the output. The depth at which the element is fetched from the FIFO must be less than the running total of newly generated references (*NEWREF*). This is achieved by normalising the random number before it is mapped to the stack distance cumulative distribution. Stack distance values are selected from the stack distance probability vector (*SD*) using its corresponding cumulative probability distribution (*F*). Theoretically, the maximum possible stack distance value is the length of the FIFO, but in practice it is the value of the last element in *SD*. Both *SD* and *F* are numerically ordered vectors as *F* is a monotonically increasing cumulative distribution function. As the trace generation progresses, the stored working-set organises itself such that the reference element at the front of the queue is the most recently used, with frequency gradually reducing up to the least recently used reference element at the other end. The procedure for arbitrary length trace generation is summarised in Listing 2. The algorithm for stack distance generation is described in Listing 3.

```
procedure TraceGen(L, SD, F, LCOUNT)
declareFIFO(S)
initialise(S, L)
SIZE:=getLength(S)
TLENGTH=arbitrary
B:=32
NEWREF:=0
    for i:=0 to TLENGTH
        sd:=genStackDistance(SD,F, NEWREF)
        if sd=-1 then
            memRef:=S[0]
            popBack(S)
            pushFront(S, memRef)
            memRef:=memRef*B
            NEWREF:=NEWREF+1
        else
            memRef:=S[SIZE-1-sd]
            memRef:=memRef*B
            pop(S, SIZE-1-sd)
            pushFront(S, memRef)
        end if
    end for
end procedure
```

Listing 2. Reference generation algorithm.

```
procedure genStackDistance(SD, F, NEWREF)
SIZE:=getLength(SD)
maxSD:=SD[SIZE-1]
ran:=randomFloat(0,1)
if NEWREF<=maxSD then
    k:=0
    while SD[k]<NEWREF
        k:=k+1
    end while
    ran:=ran*F[k-1]
end if
for k:=0 to SIZE
    if ran<F[k] then
        sd:=SD[k]
        return sd
    end if
end for
end procedure
```

Listing 3. Stack distance generation algorithm

The entire approach is briefly captured in the figure below.

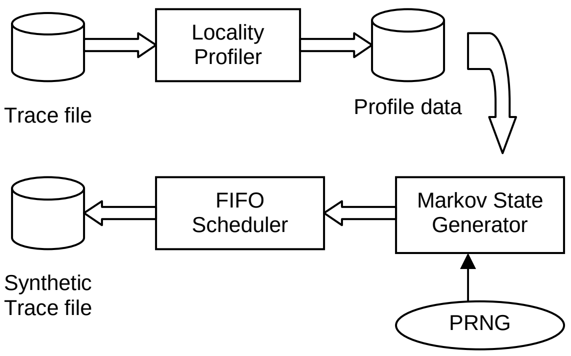

Figure 2. Synthetic reference generator.

V. Evaluation

We evaluated the approach using traces of applications from the SPEC2000 benchmark suite [19]. The applications were simulated on a Pentium III processor with the cache disabled and running Windows NT4 OS. Traces were collected using the BACH hardware monitoring technique [20]. The traces contain a billion memory references and are available from the BYU Trace Distribution Center [21]. For reasons of brevity, we present results of instruction and data performance for *crafty*, *gcc (166)*, *gzip (source)*, *parser*, *twolf* and *vortex (two)*.

A. *Trace Characterisation*

We captured the cumulative distribution of stack distance for the references traces, as illustrated in the truncated locality plots in Figure 3 and 4. The relative smoothness of the data reference curves indicates that data memory is generally referenced in a progressive manner, unlike instruction references that exhibit regular branching to procedure/function calls. Table I summarises additional features of the traces not observable in the plots: the full working-set is the number of new cache blocks observed in the program execution (using a 32-byte block size); the average *SD* is the mean number of unique cache block accesses between identical accesses and is a useful generic measure of locality:

$$\text{Average } SD = \sum_i SD_i \times P_{SD_i}$$

The distribution of *SD* for data references has a much longer tail than instruction references, as highlighted by the significantly higher average *SD*. This implies that the instruction traces exhibit a much higher degree of temporal locality, as would be expected due to modern performance enhancements such as software/hardware instruction prefetching. (Note the complete profile data typically consumes less than 0.0001% storage space of the original trace file.)

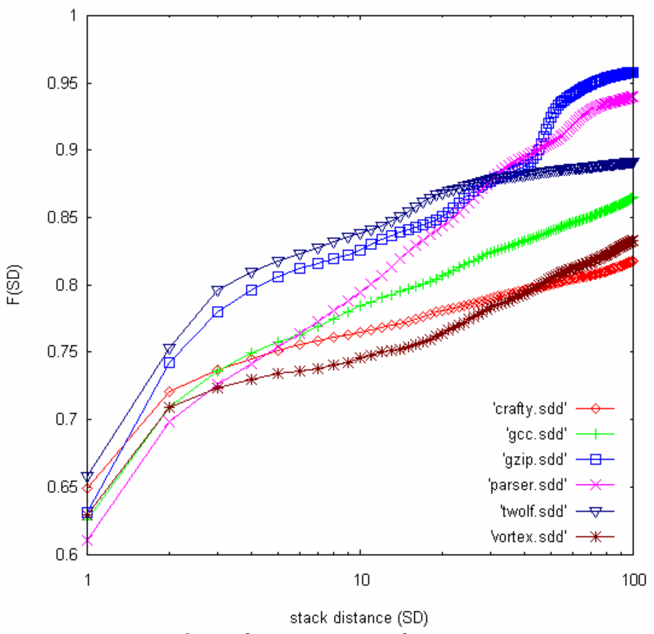

Figure 3. Locality of instruction references.

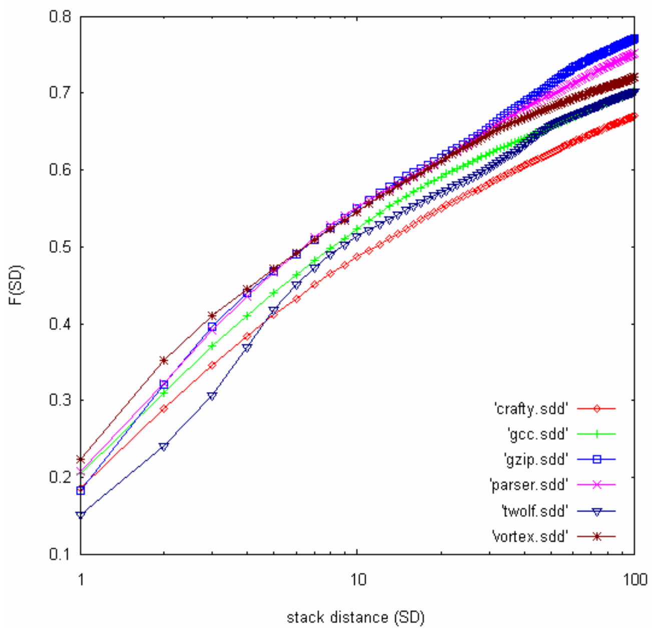

Figure 4. Locality of data references.

Table I. Instruction and data trace characteristics.

	Instruction		Data	
	Full Working-Set (blocks)	Average SD	Full Working-Set (blocks)	Average SD
crafty	54089	211.9	220105	1215.9
gcc	66083	287.9	399567	2236.9
gzip	33683	70.1	313519	1851.4
parser	24312	49.9	379782	2242.8
twolf	39522	163.9	425619	4946.9
vortex	36242	179.5	681837	1729.4

B. *Trace Simulation*

The profile data of each application was passed to the Markov state generator and driven by a pseudo-random number generator. A trace length cut-off of 10% of the original length was selected to allow the cache sufficient time to warm-up. Note, reducing trace length, as is possible in our stochastic model, highlights the potential speedup attainable in cache simulation, and is typically proportional to the level of reduction (speedup α trace length). The synthetic reference streams output from the FIFO scheduler were evaluated against their real counterpart using the DineroIII trace-driven cache simulator [4]. DineroIII was configured for Harvard cache architecture with a write-allocate write miss policy, a 32 byte cache line size and least recently used block replacement policy. We present results for 4-way and 8-way associative caches of size $\in$ {1K:128K} bytes. Figures 5 and 6 illustrate the simulation results for the real traces (*expected*) versus the synthetic forms (*observed*) plotted on a logarithmic y-axis, for instruction and data respectively.

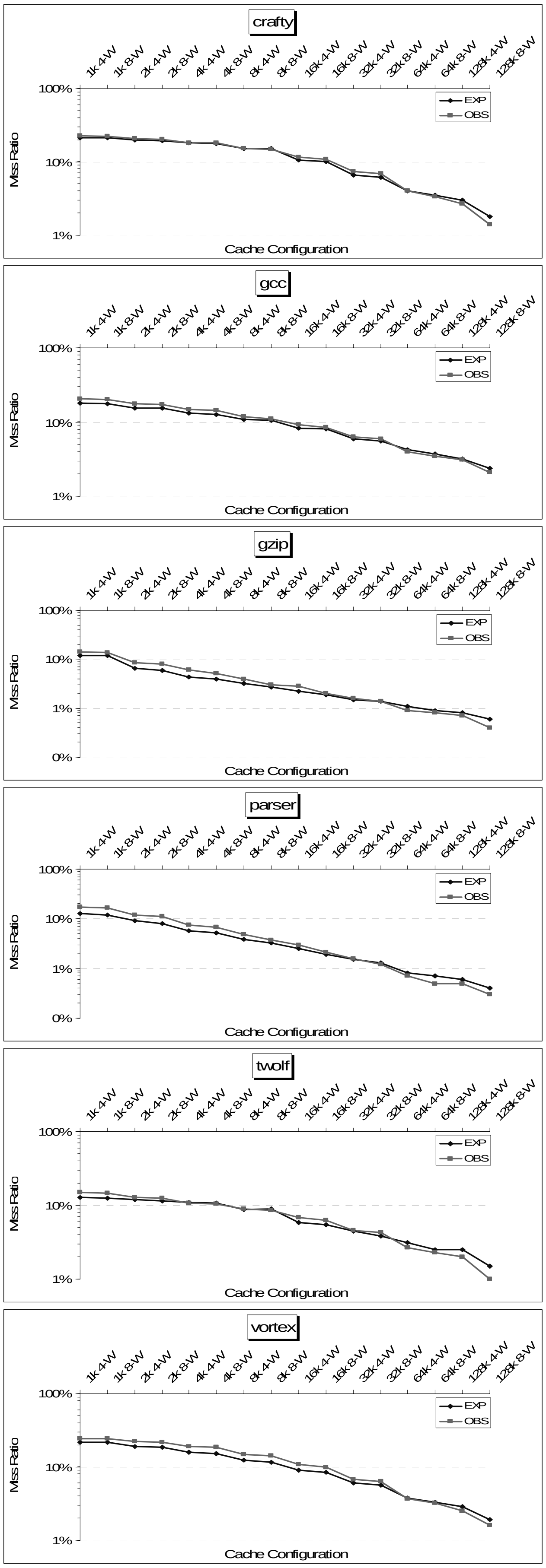

Figure 5. Instruction cache simulation results.

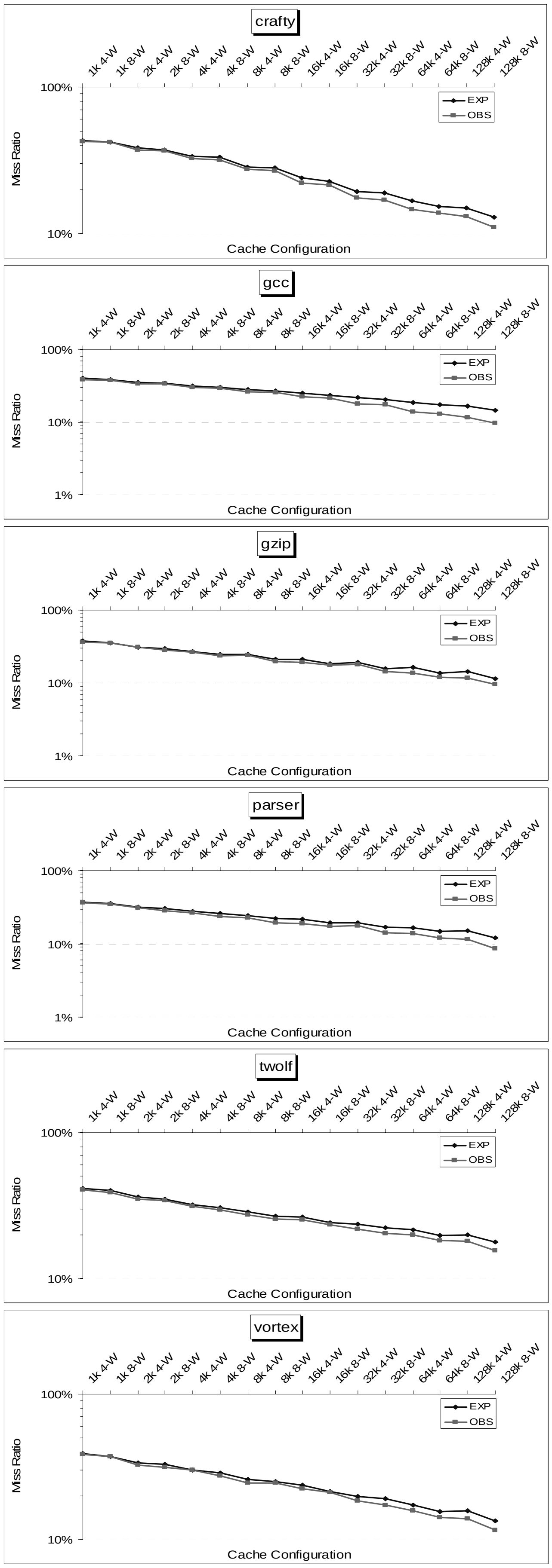

Figure 6. Data cache simulation results.

We note from the simulation results that the model performs very well in generating synthetic memory references with cacheability features very similar to the original reference stream. The model performs significantly better than existing models evaluated in the study by Sorenson [11]. Furthermore, the Markov state generator and FIFO scheduler ensure the distribution of stack distance for the synthetic stream is virtually the same as its real equivalent (in Figure 3 and 4). This indicates the reference generation procedure reproduces the temporal locality features of the original trace.

VI. Conclusions

We have presented a concise methodology for synthetic memory reference generation. An algorithm adapted from the LRUSM is used to characterise the spatio-temporal characteristics of application workloads. The profile data is passed to a Markov model which generates memory references through a dynamically ordered FIFO scheduler, and is driven by a pseudo-random number generator. Simulation of SPEC2000 traces show that the synthetic references preserve the cacheability properties of the real trace for caches operating over a range of configurations. The storage overhead is very low, while an arbitrary-length trace allows for a speedup in cache simulation. The model has applications in the area of workload generation for cache simulation and bus usage, and may also serve as the basis of a self-contained traffic generator.

References

[1] J. Connell, *ARM System-Level Modeling*, ARM Ltd, 2003.

[2] *RealView ARMulator ISS*, ARM Ltd, 2004.

[3] T. Austin et al, *SimpleScalar Tutorial v4*, University of Michigan.

[4] D. M. Hill, *DineroIII Cache Simulator*, University of California, Berkeley, 1985.

[5] D. J. Lilja, *Measuring Computer Performance: A Practitioner's Guide*, Cambridge University Press, 2000.

[6] P. Denning and S. Schwartz, *Properties of the Working-Set Model*, Communications of the ACM, 1972.

[7] J. Spirn, *Program Behavior: Models and Measurements*, Elsevier, 1977.

[8] D. Thiebaut, J. L. Wolf, and H. S. Stone, *Synthetic Traces for Trace-Driven Simulation of Cache Memories*, IEEE Transactions on Computers, 1992.

[9] L. Eeckhout, K. De Bosschere, and H. Neefs, *Performance Analysis Through Synthetic Trace Generation*, International Symposium on Performance Analysis of Systems and Software, 2000.

[10] A. Agarwal, M. Horowitz, and J. Hennessy, *An Analytical Cache Model*, ACM Transactions on Computer Systems, 1989.

[11] E. Sorenson and J. K. Flanagan, *Evaluating Synthetic Trace Models using Locality Surfaces*, International Workshop on Workload Characterization, 2002.

[12] E. Berg and E. Hagersten, *StatCache: A Probabilistic Approach to Efficient and Accurate Data Locality Analysis*, International Symposium on Performance Analysis of Systems and Software, 2004.

[13] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, *Evaluation Techniques for Storage Hierarchies*, IBM System Journal, 1970.

[14] K. Grimsrud, J. Archibald, R. Frost, and B. Nelson, *On The Accuracy of Memory Reference Models*, Seventh International Conference on Modeling Techniques and Tools for Computer Performance Evaluation, 1994.

[15] M. Brehob and R. Enbody, *An Analytical Model of Locality and Caching*, Michigan State University, 1999.

[16] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kauffman Publishers, 2003.

[17] D. Spinellis, *Code Quality: The Open Source Perspective*, Addison Wesley, 2006.

[18] Luc Devroye, *Non-Uniform Random Variate Generation*, Springer-Verlag, 1986.

[19] SPEC Benchmark Suite, http://www.spec.org.

[20] J. K. Flanagan, B. Nelson, J. Archibald, and K. Grimsrud, *BACH: BYU Address Collection Hardware*, 6th International Conference on Modeling Techniques and Tools for Computer Performance Evaluation, September 1992.

[21] BYU Performance Evaluation Laboratory, http://pel.cs.byu.edu/.