Synthetic Trace-Driven Simulation of Cache Memory

Rahman Hassan
Institute for System Level Integration, Livingston, UK.
rhassan@sli-institute.ac.uk

Antony Harris
ARM Ltd, Sheffield, UK.
aharris@arm.com

Nigel Topham, Aris Efthymiou
School of Informatics, University of Edinburgh, UK.
{npt, aefthymi}@inf.ed.ac.uk

Abstract

The widening gap between CPU and memory speed has made caches an integral feature of modern high-performance processors. The high degree of configurability of cache memory can require extensive design space exploration and is generally performed using execution-driven or trace-driven simulation. Execution-driven simulators can be highly accurate but require a detailed development flow and may impose performance costs. Trace-driven simulators are an efficient alternative but maintaining large traces can present storage and portability problems. We propose a distribution-driven trace generation methodology as an alternative to traditional execution- and trace- driven simulation. An adaptation of the Least Recently Used Stack Model is used to concisely capture the key locality features in a trace and a two-state Markov chain model is used for trace generation. Simulation and analysis of a variety of embedded application traces demonstrate the cacheability characteristics of the synthetic traces are generally very well preserved and similar to their real trace, and we also highlight the potential performance improvement over ISA emulation.

1. Introduction

Caches are highly configurable features of modern processors and their architecture is characterised by parameters such as size, associativity, line (block) size, and replacement policy. Cache performance can vary considerably depending on the choice of configuration and workload, with penalties for an incorrectly configured cache including an increase in latency and area overhead (in using oversized caches). In order to maximise performance and reduce cost, much emphasis is therefore placed on finding the optimum configuration for an expected application workload.

Cache memory is usually evaluated using execution-driven or trace-driven simulation. Execution-driven simulation can be performed at various levels of abstraction, from the algorithmic level to the 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. However, execution-driven simulation can be slow and requires architectural models, application source-code, and a development toolkit. Trace-driven simulation is a faster and increasingly common way of evaluating memory systems. Trace-driven simulators such DineroIII [4] accept a chronological stream of memory references and evaluate miss statistics based on the selected configuration. Trace-driven simulation can be an attractive way of exploring multi-level cache designs and multiprocessor system caches. However, as applications become more complex, they generate increasingly larger traces. A key problem is the storage requirement of enormous trace files containing hundreds of millions of memory references. Truncation, trace-sampling, and compression can be employed to reduce the effective length of a trace but often at the expense of accuracy and/or time. Synthetic trace generation can address the problems of maintaining large traces by using a distribution-driven (stochastic) model to generate an arbitrary number of memory references on-the-fly; the reference stream can then be passed to a trace-driven cache model for system evaluation. Unfortunately, synthetic trace models usually lack accuracy [11] unless detailed profiling procedures are employed, and as such may not always be suitable for fast and accurate cache design space exploration.

2. Related Work

An early proposal by Denning [6] considered generating memory references based on their independent probability. Known as the Independent Reference Model (IRM), the approach fails 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 uses two parameters corresponding to 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 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 does not present a practical approach to quantify locality but does evaluate 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 due to the fact that 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 author's assumption that the larger the reuse distance, the higher the probability of a cache line eviction is not necessarily true. The intermediate accesses could all be to the same memory location and a large reuse distance would not reflect this pattern. A much better measure of locality is presented by Mattson [13] in the form of the Least Recently Used Stack Model (LRUSM). The LRUSM 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 emphasise the fact that the LRUSM is a natural representation of least recently used behaviour. However, the works do not analyse the efficacy of the LRUSM in the generation of synthetic traces for trace-driven simulation of cache memory. In our work, we implement an adaptation of the LRUSM and apply it to an algorithm employing a two-state Markov chain and show we can generate accurate synthetic traces aimed at cache simulation using both least recently used and random block replacement policies. We use traces collated from application benchmarks executed on an embedded processor and evaluate the approach using a comprehensive range of cache configurations. We also compare the approach against ISA emulation to highlight the potential performance improvement.

3. Trace Locality

An important general rule of program execution is the 90/10 rule [16], also known as the *Pareto Principle* [17]. The rule states that 90% of a program's execution time is spent in only 10% of the code and highlights the significance of locality in evaluating and optimising 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. A synthetic memory reference model must seek to preserve the original temporal and spatial locality information. We capture spatial locality using line-size granularity and map memory references to cache line numbers. Any unique references mapping to the same cache line are treated as identical references. We use a distribution based on the LRUSM to quantify temporal locality. The LRUSM is formulated from the number of unique intervening

memory references between two identical references. Although intended to apply to caches with LRU block replacement, the stack distance is a useful metric of temporal locality regardless of replacement policy and can in fact form the foundation for trace generation for caches with random block replacement. Listing 1 describes the pseudo-code of the trace profiling algorithm.

```
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.

We use a dynamically growing integer stack (*S*). For each memory reference *R* we see 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 ∞ as they have not been referenced previously but we use a finite value of *-1* to enable quantitative profiling for the trace generation algorithm. We also capture new line accesses and the order in which they appear (*L*), the number of which we describe 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 to generate a distribution of probabilities. Stack distance values with a probability lower than 0.001% are discarded to reduce the size of the distribution set and to improve performance of the trace generation algorithm. The probability distribution is then integrated to generate the cumulative distribution of stack distance values as a non-decreasing function:

$$F_i=P_i+F_{i-1} \quad \forall\ i \quad where\ F_0=P_0$$

The cumulative probability distribution *F* and corresponding stack distance values *SD* are stored as numerically ordered probability vectors in separate data structures.

4. 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. Both states are governed by the stack distance cumulative probability distribution vector *F*.

Stack distance values are generated by a pseudo-random number generator issuing a number in the interval [0:1] that is mapped to a stack distance using the Inverse Transform Sampling method [25], as illustrated in Figure 1.

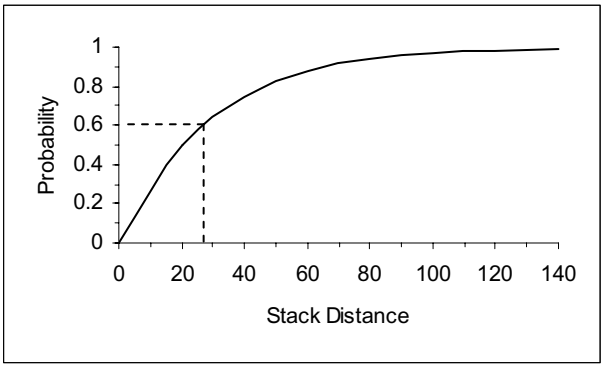


Figure 1. Stack distance mapping.

A stack distance value of *-1* issues a new memory reference while any other value generates a previous reference. Inter-state and intra-state probabilities are treated as stochastically independent in line with the Markov property, as shown in Figure 2.

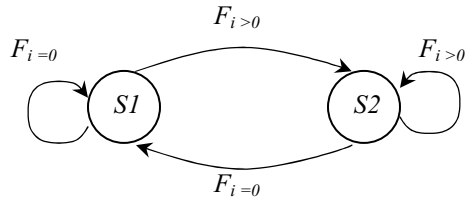


Figure 2. Markov model for trace generation.

The key to the algorithm for random block replacement caches 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 front of the FIFO is popped off and pushed to the back, 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 back 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 dynamically scaling 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). The maximum possible stack distance value in theory is the length of the FIFO, but practically 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. Listing 2 summarises the procedure for arbitrary length trace generation.

```
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]
        popFront(S)
        pushBack(S, memRef)
        memRef:=memRef*B
        NEWREF:=NEWREF+1
    else
        memRef:=S[SIZE-1-sd]
        memRef:=memRef*B
    end if
end for
end procedure
```

Listing 2. Trace generation algorithm for random replacement caches.

For LRU replacement, the procedure is almost identical except for a slight modification in the scheduler. As before, memory references are output from the top of the stack for each new reference while previous references use the bottom of the stack as the base and an offset equal to the requested stack distance. However additionally, each request for a previous reference causes the reference element at that depth to be removed and pushed to the bottom of the stack to represent the fact it was the most recently used. As the trace generation progresses, the stack organises itself such that the reference element at the bottom of the stack is the most recently used, with frequency gradually reducing up to the least recently used reference element at the top of the stack. Listing 3 summarises the procedure. Listing 4 presents the algorithm for stack distance generation employed in both procedures.

```
procedure TraceGen(L, SD, F, LCOUNT)
declareStack(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]
        popTop(S)
        pushBottom(S, memRef)
        memRef:=memRef*B
        NEWREF:=NEWREF+1
    else
        memRef:=S[SIZE-1-sd]
        memRef:=memRef*B
        pop(S, SIZE-1-sd)
        pushBottom(S, memRef)
    end if
end for
end procedure
```

Listing 3. Trace generation algorithm for LRU replacement caches.

```
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 4. Stack distance generation algorithm

5. Evaluation

We evaluated the approach using the ARMulator instruction set simulator [2, 20]. ARMulator simulates the instructions sets and architecture of a variety of ARM processors, as well as memory systems and peripherals. We selected an ARM926 processor model [18], which has a Harvard cached architecture and hosts an ARM9 32-bit integer core. It was connected to program and data memory models through separate AMBA AHB interfaces. We simulated a variety of application benchmarks that may typically run in an embedded system:

1. *mpeg2enc* – MPEG-2 format video encoder from the MediaBench benchmark suite [21].
2. *djpeg* – JPEG format image decoder from the EEMBC Consumer benchmark suite [22].
3. *aes* – security application from the EEMBC Consumer benchmark suite that implements the Advanced Encryption Standard using the Rijndael algorithm [22].
4. *wcdma* – application program that emulates the physical layer operation of the W-CDMA communications protocol [23].
5. *go* – artificial intelligence game from the SpecInt95 benchmark suite that plays the game Go against itself [24].
6. *compress* – compression algorithm from the SpecInt95 benchmark suite that employs Limpel-Ziv encoding [24].

Executable images of the application source code were created using the ARM development toolkit [19]. The source code was compiled with optimisation level `-O2` and targeted specifically for the ARM9 core to maximise use of any static scheduling and instruction-set extensions. ARM program code supports static prefetching by way of conditional code generated by the compiler, in addition to the dynamic prefetching offered by the dedicated prefetch unit in the core. For our validation, we analysed traces of data transactions initiated by the core.

5.1 Trace Characterisation

We captured the cumulative distribution of stack distance for the data references of the application benchmarks. A stack distance value of zero is a cache line repetition, or in other words an intra-line memory reference, and is the single most frequent occurrence due to the naturally sequential nature of program execution and the atomic execution of multiple load/store operations. We chose not to include line repetition in our analysis as it has no bearing on the number of cache misses. The stack distance distribution of the references is illustrated in Figure 3. The relative smoothness of the curves indicates that the data memory locations are generally referenced in a progressive, orderly manner.

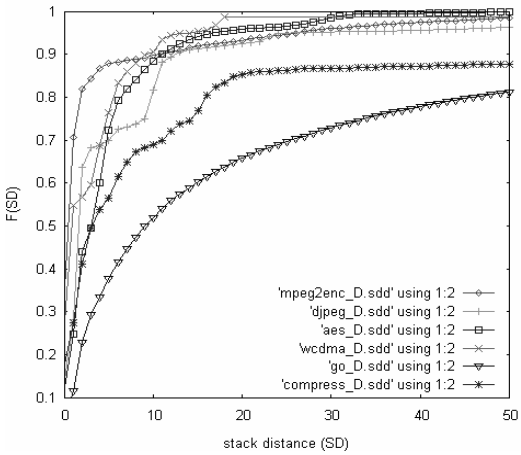


Figure 3. Cumulative distribution of stack distance for the data reference traces.

Table I summarises some of the characteristics of the data traces. The ratio of dynamic to static coverage is defined as the ratio of the number of new lines observed in the program execution (full working-set) to the number of cache lines in the static image. We use a 32-byte line size. The static data size of the image is the combined size of the read-only data (constants, literals, etc), read-write data and zero-initialised data. Additionally, its full working-set also includes stack and heap accesses. The average stack distance can be a useful basic metric of locality and is defined as the mean number of unique cache line accesses between identical accesses:

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

	Static Data Size (Bytes)	Full Working-Set (lines)	Dynamic-Static Coverage ratio	Ave. SD
1	16820	7781	46%	4.7
2	815184	28375	3.5%	73.6
3	3660	190	5.2%	5.3
4	992476	41342	4.2%	29.2
5	571468	20031	3.5%	33.7
6	44112988	1380585	3.1%	398.6

Table I. Characteristics of the traces for *mpeg2enc* (1), *djpeg* (2), *aes* (3), *wcdma* (4), *go* (5), and *compress* (6).

5.2 Trace Simulation

The profile data of each application benchmark was passed to the synthetic trace generation algorithm, which was configured to generate traces of half their original length. This was a somewhat arbitrary cut-off but allowed sufficient time for cache warmup. The synthetic traces were evaluated against their real counterpart using the DineroIII trace-driven cache simulator [4]. DineroIII was configured with a write-allocate write miss policy, and a 32 byte cache line size. In order to demonstrate the performance of the approach, we looked at set-associative caches with all ways through to full associativity and cache size $C \in$ {64:16K} bytes. Figures 4 and 5 illustrate the cache miss ratio results of the real traces (*expected*) versus the synthetic traces (*observed*) for random and LRU block replacement caches.

We note from the simulation results that the observed performance of the synthetic traces is generally representative of the expected behaviour. Although the synthetic trace for *go* consistently overestimates the miss ratio for random block replacement caches, it does so with an offset that is proportional to cache size. As a result, the synthetic trace still has the potential to perform accurately to find the best cache configuration for that application (based on some criteria such as minimising miss ratio, area, and/or latency) since the observed results do very well to track the expected miss ratios, albeit with the proportional offset. While it is accepted that no synthetic trace generation model can consistently generate exact cache simulation results for every cache configuration and for every input trace due to the very nature of stochastic modelling, our results typically show that we are able to preserve the cacheability properties of the real trace and generate a synthetic trace with similar behaviour for random and LRU block replacement caches operating over a wide range of configurations. A comparison with the results presented by Sorenson [11] demonstrates the improved accuracy of the approach relative to some existing models.

5.3 Performance Evaluation

The speed of execution-driven cache simulation can be significantly affected not only by workload characteristics but also the cache configuration. A notable trade-off exists between cache size and associativity. A smaller cache size causes a higher number of capacity misses, thereby increasing latency by the increased number of bus transactions. A higher associativity serves to reduce the number of conflict misses (and therefore bus usage), but the way selection logic imposes its own latency overhead. Using ARMulator v1.4 running on the Intel Pentium IV 3.00GHz CPU under Microsoft XP, we assessed the performance of ARMulator executing the *mpeg2enc* application benchmark for cache size C ∈ {1K, 16K} bytes and associativity A ∈ {direct-mapped:fully-associative}. The results were compared with the combined time of the synthetic trace generation algorithm and the corresponding trace-driven cache simulation in the manner described previously. Table II illustrates the results for 135×10^6 instruction executions. The table shows the performance of ISA emulation is dependent on cache configuration and that performance can deteriorate with increasing associativity and/or decreasing cache size. On the other hand, the synthetic trace generation and simulation generally takes a fixed length of time.

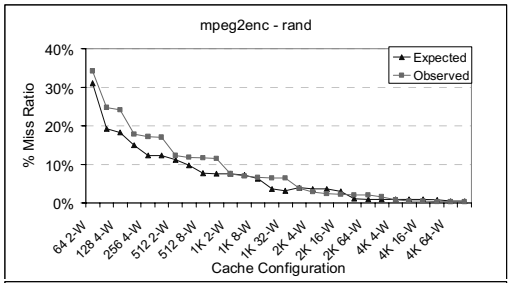
	time(secs)			
A	C=1K		C=16K	
	ISA	Model	ISA	Model
1	138	76	121	76
2	131	76	117	76
4	125	76	117	76
8	133	77	119	76
16	127	78	122	77
32	415	81	142	78
64	-	-	170	78
128	-	-	182	79
256	-	-	192	80
512	-	-	1495	82

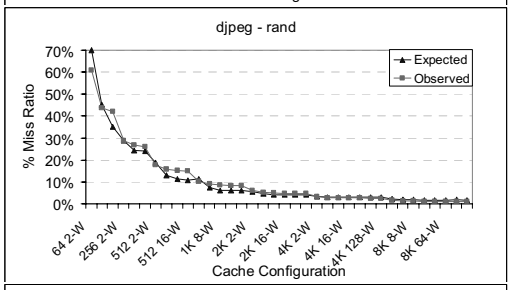
Table II. Performance evaluation results.

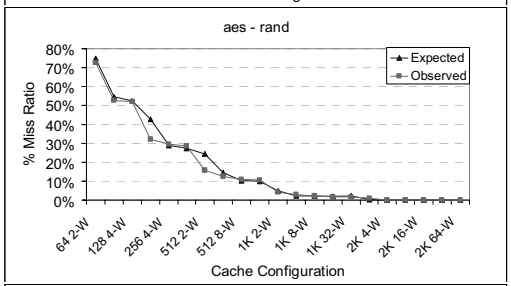
6. Conclusions

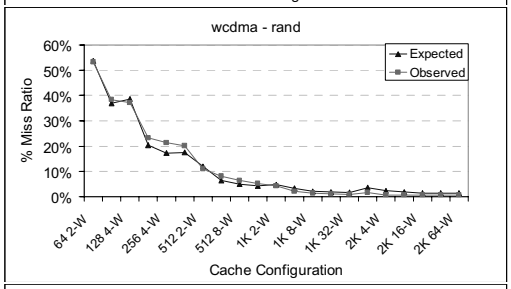
Exploration of cache design space is usually performed using execution-driven or trace-driven simulation. Achieving the accuracy of execution-driven simulation is a trade-off against performance and factors such as architectural simulation models and application source-code. Trace-driven simulation is an efficient alternative, but large traces can present significant storage and portability problems.

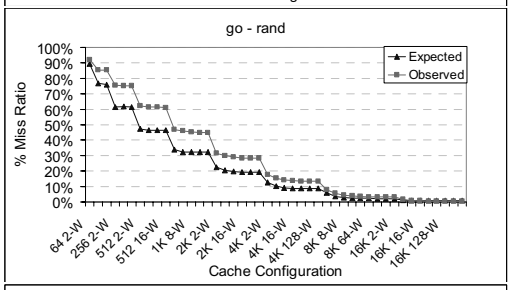
We have presented a synthetic trace generation methodology for trace-driven cache simulation that uses an efficient adaptation of the LRUSM that employs cache line profiling to concisely capture trace locality. A trace generation algorithm using a two-state Markov chain model is used to generate arbitrary length traces independent of cache size and associativity. Extensive simulation and analysis of traces of a variety of application benchmarks show the











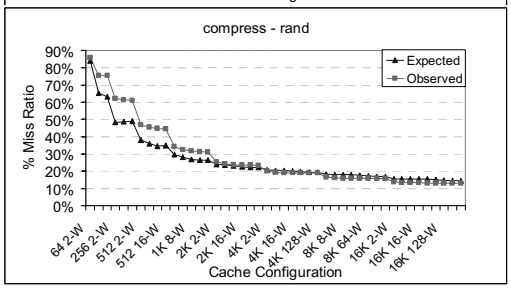
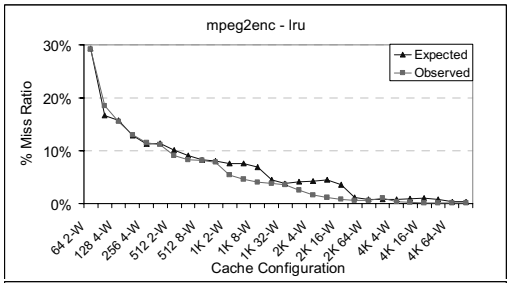
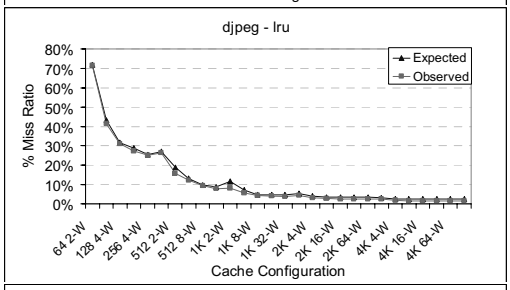
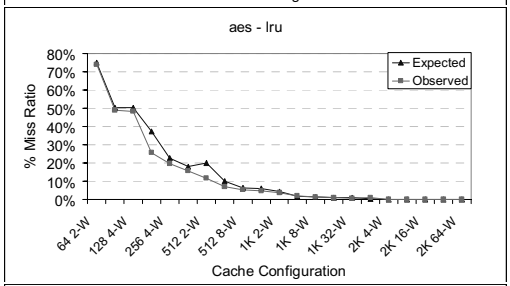
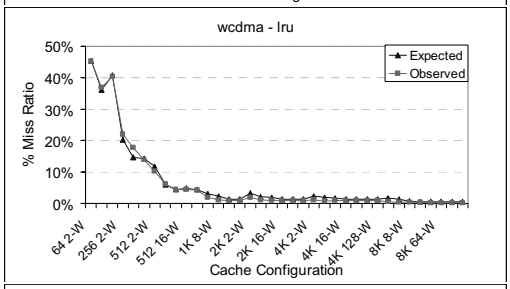


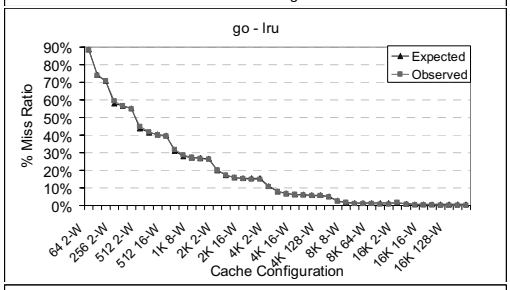
Figure 4. Results for random block replacement.











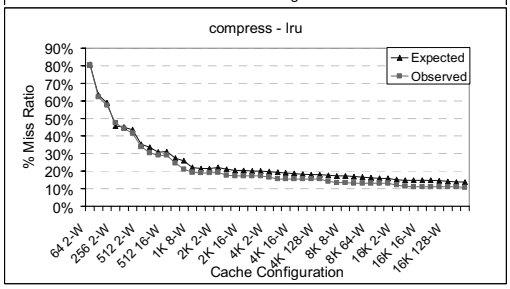


Figure 5. Results for LRU block replacement.

synthetic traces generally preserve the cacheability properties of the real trace for both LRU and random block replacement caches operating over a wide range of configurations. Performance evaluations against the ARMulator ISS show that the simulation speed of the approach is generally independent of cache architecture and has the potential to perform significantly faster than ISA emulation.

7. 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*, IEEE 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*, IEEE Workshop on Workload Characterization, 2002.

[12] E. Berg and E. Hagersten, *StatCache: A Probabilistic Approach to Efficient and Accurate Data Locality Analysis*, IEEE 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] *ARM926E-S Technical Reference Manual*, ARM Ltd, 2001

[19] *ARM Developer Suite: Compiler, Linker, and Utilities Guide*, ARM Ltd, 2000.

[20] *RealView Developer Suite: AXD and armsd Debuggers Guide*, ARM Ltd, 2004.

[21] C. Lee, M. Potkonjak, and H. Mangione-Smith, *MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems*, Micro-30, November 1997.

[22] EDN Embedded Microprocessor Benchmark Consortium, http://www.eembc.org.

[23] H. Lee, *wcdmaBench*, Software Defined Radio Group, University of Michigan, 2006.

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

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