Test data generation for LRU cache-memory testing

Evgeni Kornikhin
Moscow State University, Russia
Email: kornevgen@gmail.com

***Abstract*—System functional testing of microprocessors deals with many assembly programs of given behavior. The paper proposes new constraint-based algorithm of initial cache-memory contents generation for given behavior of assembly program (with cache misses and hits). Although algorithm works for any types of cache-memory, the paper describes algorithm in detail for basis types of cache-memory only: fully associative cache and direct mapped cache.**

I. Introduction

System functional testing of microprocessors uses many assembly programs (*test programs*). Such programs are loaded to the memory, executed, execution process is logged and analyzed. But modern processors testing requires a lot of test programs. Technical way of test program generation was proposed in [1]. This way based on the microprocessor's model. Its first stage is systematic generation abstract test programs (*test templates*). This abstract form doesn't contain initial state of microprocessor but contain sequence of instructions with arguments (registers) and with *test situations* (behavior of this instruction; these can be overflow, cache hits, cache misses). The second stage is generation of initial microprocessor state for given test template. This stage is test data generation. Technical way from [1] is useful for aimed testing when aim is expressed by instruction sequence with specific behavior. Initial microprocessor state includes initial values of registers and initial contents of cache-memory. Based on this state the third, final, stage is generation the sequence of instructions to reach initial microprocessor state. These sequence of instructions with test template get ready assembly program. This paper devoted to the second stage, i.e. initial state generation.

Known researches about test data generation problem contain the following methods of its solving:

1) combinatorial methods;
2) ATPG-based methods;
3) constraint-based methods.

Combinatorial methods are useful for simple test templates (each variable has explicit directive of its domain, each value in domain is possess) [2]. ATPG-based methods are useful for structural but not functional testing [3]. Constraint-based methods are the most promising methods. Test template is translated to the set of constraints (predicates) with variables which represented test data. Then special solver generates values for variables to satisfy all constraints. This paper contains constraint-based method also. IBM uses constraint-based method in Genesys-Pro [4]. But it works inefficiently on test templates from [1]. Authors of another constraint-based methods restrict on registers only and don't consider cache-memory.

II. Test templates description

Test template defines properties of future test program. Test template contains sequence of instructions. Each element of this sequence has instruction name, arguments (registers, addresses, values) and test situation (relation between values of arguments and microprocessor state before execution of instruction). Example of test template description for model instruction set:

```
REGISTER reg1 : 32;
REGISTER reg2 : 32;
ADD reg1, reg2, reg2
LOAD reg1, reg2 @ l1Miss, l2Hit
SUB reg2, reg1, reg2
```

This template has 3 instructions – ADD, LOAD SUB. Template begins from variable definitions (it has name of variable and its bit length). Test situation is specified after "@": test situation of the second instruction is "l1Miss, l2Hit": "l1Miss" means cache miss in first-level cache and "l2Hit" means cache hit in second-level cache.

Model instruction set contains only 2 memory operation:

- "LOAD reg, address" loads value from memory by physical address "address" to the register "reg";
- "STORE reg, address" stores value from register "reg" to the memory by physical address "address".

Test data generation is generation of initial values of registers and initial contents of cache-memory. This problem has been solved for common microprocessor cache-memory. The following consists of test data generation for 2 basis cache-memory organizations: fully associative cache with LRU and direct mapped cache. Common cache includes aspects from both cache-memory organizations. The rest of paper deals with one-level cache-memory although proposed method can be applied to cache memory with more than one level.

III. Test data generation for fully associative cache

Fully N-associative cache consists of N cells (N means *cache associativity*). Each cache cell may store data from any memory cell. All cache cells correspond to the different memory cells. Access to memory starts from access to cache. Search data in cache performs for each cache cells in parallel. *Cache hit* means existence data in cache. *Cache miss* means absence of data in cache. In case of cache miss one cache

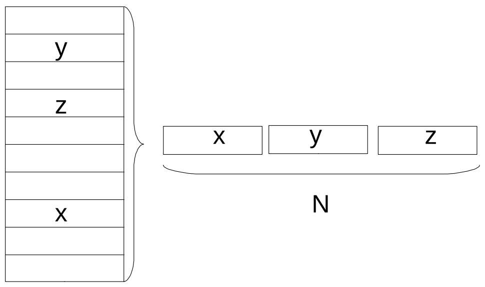

Fig. 1. Fully N-associative cache

cell must be replaced on data from required address by specific *replacement strategy*. This paper uses LRU replacement strategy (Least Recently Used). According to LRU the least recently used cache cell will be evicted. At the following phrase ”evicted address x” means evicted data by address x.

Proposed algorithm based on the following properties of evicted addresses:

1) any evicted address was inserted by instruction from test template with cache miss or was in the initial contents of cache;
2) between replacing and the last access to the same address (cache hit or cache miss) there are accesses to the whole cache without address itself.

Proposed algorithm generates constraints on the following variables:

1) $\alpha_1, \alpha_2, ..., \alpha_N$ – initial contents of cache (its count equals to cache associativity);
2) hits-addresses (addresses of instructions from test templates with cache hit test situation);
3) misses-addresses (addresses of instructions from test templates with cache miss test situation);
4) evicted addresses (evicted addresses of instructions from test templates with cache miss test situation);
5) $L_0, L_1, ...$ – cache states

Each instruction from test template with cache hit gives 1 new variable, and each instruction with cache miss gives 3 new variable (1 for miss address, 1 for evicted address, and 1 for cache state). Proposed algorithm generates constraints for each instruction from test template by the following (N means cache associativity):

1) ”initial constraints” are generated one time for any test template: $L_0 = \{\alpha_1, \alpha_2, ..., \alpha_N\}$, $|L_0| = N$ (other words, numbers $\alpha_1, \alpha_2, ..., \alpha_N$ are different);
2) ”hit-constraints” are generated for each instruction from test template with cache hit: $x \in L$, when x means address from instruction, L means a current cache state-variable;
3) ”miss-constraints” are generated for each instruction from test template with cache miss (x means evicting address, y means evicted address, L means a current cache state-variable): $y \in L, x \notin L, L' = L \cup \{x\} \setminus \{y\}, lru(y)$, L' became a current cache state-variable for the next instruction.

Constraint $lru(y)$ defines y as the least recently used address.

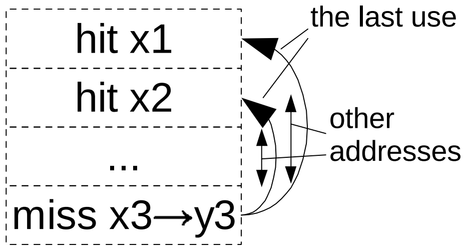

Fig. 2. LRU

Constraint $lru(y)$ is disjunction of constraints corresponded to cases of the last access to the y before its eviction. Each its clause is conjunction of the following constraints (x means the address-variable from the last access to the y):

1) $x = y$
2) $L \setminus \{y\} = \{x_1, x_2, ..., x_n\}$, where $x_1, x_2, ..., x_n$ are all addresses accessed between accesses to x and y (hits and misses).

The last access to the y can correspond to the previous instruction of test template or to the cell from initial cache state.

Consider an example of test template and its test data generation for 3-associative cache.

LOAD x, y @ Hit
STORE u, z @ Miss
LOAD z, y @ Hit

Define unique names for variables in test template (each new variable shouldn’t change its value). LOAD gives new version for its first argument. STORE doesn’t generate new version of variables. Define new variable z'_0 for evicted address from the second instruction (this variable won’t be included to the solution):

LOAD x_1, y_0 @ Hit
STORE u_0, z_0 @ Miss $\to z'_0$
LOAD z_1, y_0 @ Hit

Define variables for initial contents of cache: $\{\alpha, \beta, \gamma\}$ (its count equals to cache associativity).

So the task is looking for values of $x_0, y_0, z_0, u_0, \alpha, \beta, \gamma$ according to test template. This task has more than 1 solutions. But any solution is enough.

The first constraints describe cache hits and misses as belong to the current state of cache:

$y_0 \in \{\alpha, \beta, \gamma\}$,
$z_0 \notin \{\alpha, \beta, \gamma\}$,
$z'_0 \in \{\alpha, \beta, \gamma\}$,
$y_0 \in \{\alpha, \beta, \gamma\} \setminus \{z'_0\} \cup \{z_0\}$,
α, β, γ – different

Define constraint $lru(z'_0)$. Candidates of the last access to the this address are $y_0, \gamma, \beta, \alpha$. The first and the second candidates aren’t suitable because constraint $L \setminus \{z'_0\} = X$ is false because of different compared sets capacity. Remainder candidates give the following disjunction:

$z_0' = \beta \wedge \{\alpha, \beta, \gamma\} \setminus \{z_0'\} = \{\gamma, y_0\}$
$\vee$
$z_0' = \alpha \wedge \{\alpha, \beta, \gamma\} \setminus \{z_0'\} = \{\beta, \gamma, y_0\}$
Simplify it:
$z_0' = \beta \wedge \{\alpha, \gamma\} = \{\gamma, y_0\}$
$\vee$
$z_0' = \alpha \wedge \{\beta, \gamma\} = \{\beta, \gamma, y_0\}$
Further simplify:
$z_0' = \beta \wedge y_0 = \alpha$
$\vee$
$z_0' = \alpha \wedge y_0 \in \{\beta, \gamma\}$
Consider the first clause with the rest of constraints (variable z_0' isn't needed in solution):
$y_0 = \alpha$
$z_0 \notin \{\alpha, \beta, \gamma\}$,
α, β, γ – different

Note that x_0 and u_0 don't take part in constraints. So their values may be arbitrary.

Lets bit length of addresses is 8. So domain of all variable-addresses is from 0 to 255. Satisfying constraints variables can get the following values (these values are not unique):
$\alpha = y_0 = x_0 = u_0 = 0$
$\beta = 1$
$\gamma = 2$
$z_0 = 3$

Verify test template execution with computed initial cache state and register values:
initial cache state is [2, 1, 0]
LOAD x, 0 - Hit, because $0 \in \{2, 1, 0\}$; according to LRU the next cache state is [0, 2, 1]
STORE 0, 3 - Miss, because $3 \notin \{0, 2, 1\}$; according to LRU 3 goes to cache, 1 is evicted from cache, the next cache state is [3, 0, 2]
LOAD z, 0 - Hit, because $0 \in \{3, 0, 2\}$
All instructions from test template were executed according to given test situations.

IV. Test data generation for direct mapped cache

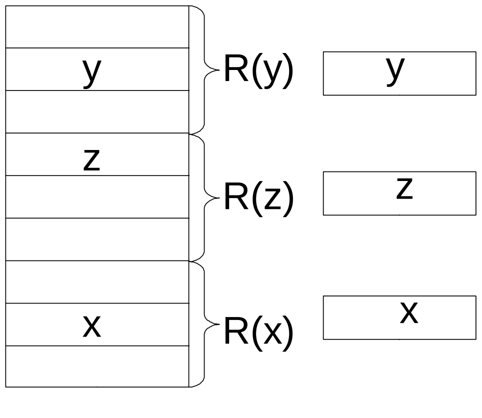

Fig. 3. Direct mapped cache

Whole memory is divided into non-intersecting areas (*regions*). Direct mapped cache consists of 1 cell for each region. Each cache cell may store data only from its region. Access to memory starts from access to cache. *Cache hit* means successful match cached address with required address in its region. *Cache miss* means unsuccessful match cached address with required address in its region. In this case data from cache replaced by data from memory by required address.

Proposed algorithm generates constraints on the following variables:

1) $\alpha_1, \alpha_2, \alpha_3, ...$ are addresses of the initial cache state (their count is regions' count);
2) hits-addresses (addresses of instructions from test templates with cache hit test situation);
3) misses-addresses (addresses of instructions from test templates with cache miss test situation);
4) evicted addresses (evicted addresses of instructions from test templates with cache miss test situation);
5) $L_0, L_1, ...$ – cache states

Define function $R(y)$ which for address y returns a set of all cells from the same region as region of y. R satisfies the following properties:
$\forall x \ (x \in R(x))$
$\forall x \ \forall y \ (x = y \rightarrow R(x) = R(y))$
$\forall x \ \forall y \ (R(x) = R(y) \leftrightarrow x \in R(y))$
$\forall x \ \forall y \ (R(x) = R(y) \leftrightarrow y \in R(x))$
$\forall x \ \forall y \ (x \notin R(y) \rightarrow x \neq y)$

Proposed algorithm generates constraints for each instruction by the following way (N means number of regions):

1) "initial constraints" are generated one time for each template : $|\{\alpha_1, \alpha_2, ..., \alpha_N\}| = N$ (other words, numbers $\alpha_1, \alpha_2, ..., \alpha_N$ are different), $|\{R(\alpha_1), R(\alpha_2), ..., R(\alpha_N)\}| = N$ (other words, all sets $R(\alpha_1), R(\alpha_2), ..., R(\alpha_N)$ are different);
2) "hits-constraints" are generated for each instruction with cache hit: $x \in L$, where x means address from instruction, L means a current variable-state of cache memory;
3) "miss-constraints" are generated for each instruction with cache miss (x means evicting address, y means evicted address, L means a current variable-state of cache): $y \in L, x \notin L, L' = L \cup \{x\} \setminus \{y\}, R(y) = R(x)$, L' became the current variable-cache state for the next instruction.

Constraints for direct mapped cache differ from constraints for fully associative cache by evicted address constraints only.

Consider test data generation for the already known test template. Lets memory divided into 3 regions depended on remainder from division address to 3 (i.e. $R(x) = R(y) \Leftrightarrow 3 | (x - y)$).
LOAD x, y @ Hit
STORE u, z @ Miss
LOAD z, y @ Hit

Define unique names for variables in test template (each new variable shouldn't change its value). LOAD gives new version for its first argument. STORE doesn't generate new version of variables. Define new variable z_0' for evicted address from the second instruction (this variable won't be included to the solution):

LOAD x_1, y_0 @ Hit
STORE u_0, z_0 @ Miss $\rightarrow z'_0$
LOAD z_1, y_0 @ Hit

Define variables of initial cache state: $\{\alpha, \beta, \gamma\}$ (one for each region).

So the task is looking for values of $x_0, y_0, z_0, u_0, \alpha, \beta, \gamma$ according to test template. This task has more than 1 solutions. But any solution is enough.

The first constraints describe cache hits and misses as belong to the current state of cache:

$y_0 \in \{\alpha, \beta, \gamma\}$,
$z_0 \notin \{\alpha, \beta, \gamma\}$,
$z'_0 \in \{\alpha, \beta, \gamma\}$,
$y_0 \in \{\alpha, \beta, \gamma\} \setminus \{z'_0\} \cup \{z_0\}$,
$R(z_0) = R(z'_0)$,
α, β, γ – different
$R(\alpha), R(\beta), R(\gamma)$ – different

Simplify this constraints set:

$z'_0 \in \{\alpha, \beta, \gamma\}$,
$y_0 \in \{\alpha, \beta, \gamma\} \setminus \{z'_0\}$,
$z_0 \notin \{\alpha, \beta, \gamma\}$,
$3|(z_0 - z'_0)$,
α, β, γ – different
$R(\alpha), R(\beta), R(\gamma)$ – different

Note that x_0 and u_0 don't take part in constraints. So their values may be arbitrary.

Lets bit length of addresses is 8. So domain of all variable-addresses is from 0 to 255. Satisfying constraints variables can get the following values (these values are not unique):

$\alpha = x_0 = u_0 = 0$
$\beta = y_0 = 1$
$\gamma = 2$
$z_0 = 3$

Verify test template execution with generated initial cache state and register values:

initial cache state is $L = [(R = 0) \mapsto 0, (R = 1) \mapsto 1, (R = 2) \mapsto 2]$

LOAD x, 1 - Hit, because $R(1) = L[R = (1 \; mod \; 3)]$

STORE 0, 3 - Miss, because $R(3) \neq L[R = (3 \; mod \; 3)]$, 1 is evicted from cache, the next state of cache is $L = [(R = 0) \mapsto 0, (R = 1) \mapsto 3, (R = 2) \mapsto 2]$

LOAD z, 0 - Hit, because $R(0) = L[R = (0 \; mod \; 3)]$

All instructions from test template were executed according to given test situations.

V. Test data generation for common cache

This section consists of illustration only the constraints for common cache.

Define function $R(x)$ as the same as for direct mapped cache.

Consider known test template for memory consisted of 3 regions ($R(x) = R(y) \leftrightarrow 3|(x - y)$) of 2-associative cache:

LOAD x, y @ Hit
STORE u, z @ Miss
LOAD z, y @ Hit

Define unique variables (and z'_0 for evicted address):

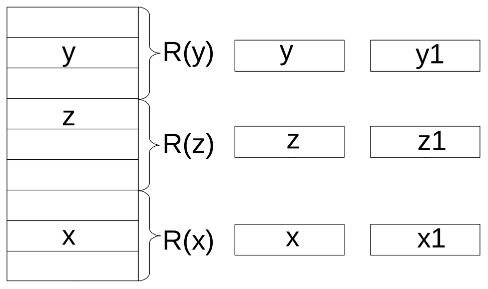

Fig. 4. Common cache

LOAD x_1, y_0 @ Hit
STORE u_0, z_0 @ Miss $\rightarrow z'_0$
LOAD z_1, y_0 @ Hit

Define variables for initial cache state: α_1, α_2 for the first region, β_1, β_2 for the second region, γ_1, γ_2 for the third region. Constraints set is the following:

$y_0 \in \{\alpha_1, \alpha_2, \beta_1, \beta_2, \gamma_1, \gamma_2\}$,
$z'_0 \in \{\alpha_1, \alpha_2, \beta_1, \beta_2, \gamma_1, \gamma_2\}$,
$z_0 \notin \{\alpha_1, \alpha_2, \beta_1, \beta_2, \gamma_1, \gamma_2\} \cap R(z'_0)$,
$y_0 \in \{\alpha_1, \alpha_2, \beta_1, \beta_2, \gamma_1, \gamma_2\} \cup \{z_0\} \setminus \{z'_0\}$,
$R(z_0) = R(z'_0)$,
$\alpha_1, \alpha_2, \beta_1, \beta_2, \gamma_1, \gamma_2$ – different,
$R(\alpha_1) = R(\alpha_2)$,
$R(\beta_1) = R(\beta_2)$,
$R(\gamma_1) = R(\gamma_2)$,
$R(\alpha_1), R(\beta_1), R(\gamma_1)$ – different

From disjunction for $lru(z'_0)$ (one clause is enough):

$z'_0 = \gamma_2 \wedge (\{\alpha_1, \alpha_2, \beta_1, \beta_2, \gamma_1, \gamma_2\} \setminus \{z'_0\}) \cap R(z'_0) = \{y_0\} \cap R(z'_0)$
$\vee$
...

Simplify:

$y_0 \in \{\alpha_1, ..., \gamma_2\}$,
$z'_0 \in \{\alpha_1, ..., \gamma_2\}$,
$z_0 \notin \{\alpha_1, ..., \gamma_2\} \cap R(z'_0)$,
$y_0 \in \{\alpha_1, ..., \gamma_2, z_0\} \setminus \{z'_0\}$,
$R(z_0) = R(z'_0)$,
$z'_0 = \gamma_2$,
$\{\gamma_1\} = \{y_0\} \cap R(\gamma_2)$
$\alpha_1, \alpha_2, \beta_1, \beta_2, \gamma_1, \gamma_2$ – different,
$R(\alpha_1) = R(\alpha_2)$,
$R(\beta_1) = R(\beta_2)$,
$R(\gamma_1) = R(\gamma_2)$,
$R(\alpha_1), R(\beta_1), R(\gamma_1)$ – different

Further simplify:

$z'_0 = \gamma_2$,
$y_0 = \gamma_1$,
$z_0 \notin \{\gamma_1, \gamma_2\}$,
$R(z_0) = R(\gamma_2)$,
$\alpha_1, \alpha_2, \beta_1, \beta_2, \gamma_1, \gamma_2$ – different,
$R(\alpha_1) = R(\alpha_2)$,
$R(\beta_1) = R(\beta_2)$,
$R(\gamma_1) = R(\gamma_2)$,
$R(\alpha_1), R(\beta_1), R(\gamma_1)$ – different

Lets bit length of addresses is 8. So domain of all variable-addresses is from 0 to 255. Satisfying constraints variables can get the following values (these values are not unique):

$\alpha_1 = 0, \alpha_2 = 3$,
$\beta_1 = 1, \beta_2 = 4$,
$\gamma_1 = 2, \gamma_2 = 5$,
$x_0 = 0, y_0 = 2, z_0 = 7, u_0 = 0$.

Special algorithms can be used for solving constraints set. These algorithms can take into account the following aspects:

- constraints can be solved symbolically;
- all sets of addresses are finite and subset of all initial cache state addresses union with evicting addresses.

VI. Conclusion

The paper devoted to the test data generation problem. Test data contains initial contents of cache-memory. The paper has proposed the constraint-based algorithm. Constraints consists of finite sets variables and sets operations. Test data generation for fully associative cache and direct mapped cache has been considered in details. Proposed algorithm is used in projects of testing MIPS-compatible microprocessors. ECLiPSe is used as constraint solver.

References

[1] A.S. Kamkin, *Test program generation for microprocessors* // Proceedings of ISP RAS. Vol. 14(2). P.23-64. 2008.

[2] K. Takayama, F. Fallah, *A new functional test program generation methodology* // Proceedings 2001 IEEE International Conference on Computer Design: VLSI in Computers and Processors. P.7681. 2001.

[3] F. Ferrandi, D. Sciuto, M. Beardo, F. Bruschi, *An approach to functional testing of vliw architectures* // Proceedings of the IEEE International High-Level Validation and Test Workshop (HLDVT00). P.2933. 2000.

[4] Y. Lichtenstein, M. Rimon, M. Vinov, M. Behm, J. Ludden, *Industrial experience with test generation languages for processor verification* // Proceedings of the 41st Design Automation Conference (DAC04). 2004.