MarciaTesta: An Automatic Generator of Test Programs for Microprocessors' Data Caches

Original
MarciaTesta: An Automatic Generator of Test Programs for Microprocessors' Data Caches / DI CARLO, Stefano; Gambardella, Giulio; Indaco, Marco; Rolfo, Daniele; Prinetto, Paolo Ernesto. - STAMPA. - (2011), pp. 401-406. (Intervento presentato al convegno IEEE 20th Asian Test Symposium (ATS) tenutosi a New Delhi, IN nel 20-23 Nov. 2011) [10.1109/ATS.2011.78].

Availability:
This version is available at: 11583/2460563 since:

Publisher:
IEEE Computer Society

Published
DOI:10.1109/ATS.2011.78

Terms of use:
This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright

(Article begins on next page)
MarciaTesta: An Automatic Generator of Test Programs for Microprocessors' Data Caches

Authors: Di Carlo S., Gambardella G., Indaco M., Rolfo D., Prinetto P.,

Published in the Proceedings of the IEEE 20th Asian Test Symposium (ATS), 20-23 Nov. 2011, New Delhi, IN.

N.B. This is a copy of the ACCEPTED version of the manuscript. The final PUBLISHED manuscript is available on IEEE Xplore®:

URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6114763

DOI: 10.1109/ATS.2011.78

© 2011 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.
mechanisms. The paper is organized as follows: Section II describes the SBST methodology to adapt march test for cache memories. Section III describes the MarciaTesta tool, exploring its main features, input/outputs, and internal architecture. Section IV shows experimental results gathered targeting the Xilinx Microblaze and the Altera Nios II microprocessors, respectively. Section V concludes the paper.

II. SBST METHODOLOGY

This section shortly overviews the SBST methodology proposed in [11] and exploited in this paper.

Traditional March tests must be properly translated before they can be applied to a cache memory. The overall translation process aims at defining basic cache march test operations that take into account the peculiar way in which cache memory cells can be accessed. The following basic operation are defined:

- \( w(\alpha, DB) \): represents a write operation of a cache line. DB is the data background pattern (for each DB a complemented DB must be defined) written in the data array section of the cache. \( \alpha \) identifies the set in which the line must be written and \( t \) identifies the tag written in the directory array of cache.
- \( r(\alpha, DB) \): represents a Read & Verify operation. The cache line placed in the set \( \alpha \) and identified by the tag \( t \) is read and compared with the expected DB.
- \( r(\alpha) \): similar to the Read & Verify operation but the read value is not verified.

We will not discuss the complex addressing order translations presented in [11] since, in its current implementation, our generation tools deals with direct-mapped data cache memories whose cache lines are directly addressable.

According to the internal cache organization, the test approaches for data and directory array are different. In data array testing, DB patterns are crucial while actual tag values are not relevant. In this context, tags are used to correctly address the cache, and the only requirement is the number of different tags. For directory array, tags became the actual test patterns. In addition, read and write operations can only be performed in an indirect way exploiting the data array. Taking into account these issues, the read verification can only be performed by detecting cache-miss events. This in turns requires creating an inconsistency between the content of the cache and that of the main memory, using different approaches based on the adopted write policy. The write-through policy requires to disable the cache in order to write a different value in the main memory, while for write-back cache policy the following rules are enough to create the miss condition:

- any write operation must be followed by another one with complemented DB;
- each Read & Verify operation must contain the value of the expected DB stored in the cache memory;
- an initialization march element must be added.

III. MARCIA TESTA TOOL

MarciaTesta tool is a general purpose tool able to generate the assembler algorithm (ASM) that implements a chosen March Test for a target processor data cache. In the sequel, we focus first on a tool overview, then we present its basic principles of operations. The section concludes with a detailed analysis of the tool’s internal architecture.

A. Input Data

The tool gets the following input data (Fig. 1):

(i) Target memory configuration

It is the input file that describes the architecture of the target data cache, according to the following parameters:

- Write policy: the write policy of the cache (Write Through or Write Back);
- Word size: the data-width of the target system.
- #Offset (O): The number of less significant bits of the memory address (O) that identify a specific word into the cache line (Fig. 2):

\[ O = \lceil \log_2(nW) \rceil \]

where \( nW \) identifies the number of words per cache line.

- #Index (I): the set of bits of the memory address that identifies the cache line where the desired information can be stored (Fig. 2):

\[ I = \lceil \log_2(nC) \rceil \]

where \( nC \) identifies the number of cache lines.

- #Tag (T): the number of most significant bits of the memory address that identify the content of the directory array used to tag the cached information (see Fig. 2):

\[ T = N - I - O \]

where \( N \) is the memory address-width.

Resorting to the values of the above presented parameters, one can easily calculate the memory’s and cache’s dimensions:

\[ \text{Cache Dim} = 2^{I+O} \]
\[ Mem_{\text{Dim}} = 2^{T+I+O} \]

- **Base address (BA):** the address serving as a reference point (“base”) for other addresses in data array testing. An important condition on this parameter is:
  \[ BA + 2 \times (T + I + O) \in \{ \text{CacheableMemory} \} \]

(ii) **March Test description**
It includes:
- The desired March Test to be implemented
- The **Addressing Order (AO)**, i.e., the exact sequence in which the cache lines must be addressed during the ascending (‘U’) and descending (‘D’) addressing orders. Such a facility is of primary importance to define custom addressing sequences required to detect dynamic memory faults [7].
- The **Data Background (DB) List**, i.e., the set of background patterns required for the implementation of the data and directory array test.

(iii) **Target processor ISA description**
It is a library that contains the minimum set of supported target microprocessors instructions, grouped in Meta-ISAs [16], useful to synthesize march test, suitable for data cache memories, in assembly code.

B. **Output Data**
The tool generates the following output data:
- **Target test program (C)**
  It is an optional output useful for test engineer for emulating test execution. It represents the C/C++ intermediate implementation of the March Test generated according to the SBST methodology.
- **Target test program (ASM)**
  It is the generated assembly code for testing the data cache.

C. **Internal architecture**
The assembly test program generation relies on two main translation steps (Fig. 3). The former one, implemented by the **YAUF2C** module, transforms the input March Test into a suitable test for data cache, exploiting a given SBST methodology. In the generated output, the test is expressed in terms of a set of C-based macros.

To better understand the translation process implemented by **YAUF2C**, in the sequel we provide three examples of
March Test translation targeting the data and directory arrays of no write-allocate caches, considering both write-through and write-back policies.

The cache write operation is implemented by means of a read operation that fills a given cache line assuming that the cache is initially empty. To guarantee this condition, all cache lines must be initially invalidated. Table II shows the translation of a test for the data array. The *Init Memory* operation is required to initialize the memory with the correct DB patterns. The other two operations show how to translate Write and Read operations, respectively. The second one is an upward r1, implemented by a sequence of invalidate operations on each cache line, followed by read memory statements at the addresses corresponding to the ascending sequence defined by the *addressing order* content.

The third operation is a downward r1 implemented by a sequence of Read & Verify operations at the addresses corresponding to the descending sequence defined by the *addressing order* content.

### TABLE II

**TRANSLATION EXAMPLE FOR DATA ARRAY TEST**

<table>
<thead>
<tr>
<th>MT</th>
<th>Target test program C</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Init Memory</em> ⇒</td>
<td>for(i=0;i&lt;pow(2,i+0);i++)</td>
</tr>
<tr>
<td></td>
<td>Write_memory_cell(DB, BaseAddress+i)</td>
</tr>
<tr>
<td></td>
<td>for(i=pow(2,i+1);i&lt;pow(2,i+1+1);i++)</td>
</tr>
<tr>
<td></td>
<td>Write_memory_cell(DB, BaseAddress+i)</td>
</tr>
<tr>
<td><em>U(w0)</em> ⇒</td>
<td>for(i=0;i&lt;pow(2,i+0);i++)</td>
</tr>
<tr>
<td></td>
<td>Invalidate_cache_line(i)</td>
</tr>
<tr>
<td></td>
<td>for(i=pow(2,i+1);i&lt;pow(2,i+2);i++)</td>
</tr>
<tr>
<td></td>
<td>Read_memory_cell</td>
</tr>
<tr>
<td></td>
<td>(BaseAddress+i,Addressing_Order[i])</td>
</tr>
<tr>
<td><em>D(r1)</em> ⇒</td>
<td>for(i=pow(2,i+1);i&lt;0;i--)</td>
</tr>
<tr>
<td></td>
<td>Read_and_verify_for_Data</td>
</tr>
<tr>
<td></td>
<td>(BaseAddress+i,Addressing_Order[i])</td>
</tr>
</tbody>
</table>

Table III and IV show a translation example of the directory array test for write-through and write-back policies, respectively.

The first row was the data background pattern at the addresses that have the T MSB equals to the desired \( DB\) TAG in order to achieve the goal to write the correct DB in the directory array when we will read at that address, knowing the corresponding pattern in the data array. In this example, the only difference between the two types of cache is in the mismatch creation between cache and memory during write operations [11]. In the write-back cache such a difference is created using \( DB\_zero\) and \( DB\_one\), while in the write-through cache it is created using Disable_Cache and Enable_Cache.

The second row presents the steps required to write in the array the desired data and to be able to read it indirectly. The first step invalidates each line of the cache, avoiding undesired cache hit during the write operation. Then the test, in the write-through case (Table III), reads the cache at the correct address \( DB\_TAG\) one), that corresponds to a \( DB\_TAG\) write in directory array [11]. After the data cache disabling, the algorithm writes, at the same address in memory, the complemented DB for the data array, creating a cache incoherence; the data cache is then enabled again. A Read & Verify operation in directory array can thus be implemented as a read from data cache, verifying that the read data equals the one written when the cache was enabled. If the read data is the second one (written when the cache was disabled), an error occurred on the directory array.

In the write-back case (Table IV), the cache incoherence is obtained by writing in the data array the complemented value of the previous written one, thus avoiding to use the disable and enable cache operations [11]. Resorting to this kind of write operation, the Read & Verify operation for the directory array can be implemented by a read from the data array, verifying that the data it is equal to the last written, \( DB\_one\) for r1 and \( DB\_zero\) for a r0. If the read data is equal to \( DB\_one\) or \( DB\_zero\), an error occurred in the directory array.

### TABLE III

**TRANSLATION EXAMPLE DIRECTORY ARRAY TEST OF A WRITE-THROUGH CACHE**

<table>
<thead>
<tr>
<th>MT</th>
<th>Target test program C</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Init Memory</em> ⇒</td>
<td>for(i=0;i&lt;pow(2,i+0);i++)</td>
</tr>
<tr>
<td></td>
<td>Write_memory_cell(DB, DB_TAG+1+i)</td>
</tr>
<tr>
<td></td>
<td>for(i=pow(2,i+1);i&lt;pow(2,i+1+1);i++)</td>
</tr>
<tr>
<td></td>
<td>Write_memory_cell(DB, DB_TAG+1+i)</td>
</tr>
<tr>
<td><em>U(w0)</em> ⇒</td>
<td>for(i=0;i&lt;pow(2,i+0);i++)</td>
</tr>
<tr>
<td></td>
<td>Invalidate_cache_line(i)</td>
</tr>
<tr>
<td></td>
<td>for(i=pow(2,i+1);i&lt;pow(2,i+2);i++)</td>
</tr>
<tr>
<td></td>
<td>Read_memory_cell</td>
</tr>
<tr>
<td></td>
<td>(DB_TAG+Addressing_Order[i])</td>
</tr>
<tr>
<td><em>D(r1)</em> ⇒</td>
<td>for(i=pow(2,i+1);i&lt;0;i--)</td>
</tr>
<tr>
<td></td>
<td>Read_and_verify_for_Directory</td>
</tr>
<tr>
<td></td>
<td>(DB_TAG,DB)</td>
</tr>
</tbody>
</table>

### C2ASM

The second main module of MarciaTesta (Fig. 3) translates the C-like March Test generated by the previous module into an equivalent assembly program exploiting the ISA of the target processor. In order to write a correct \( ASM\) software, the tool exploits the description of the ISA contained in the *Processor ISA Library* and translates the macro-operations of the C-like MT (with related parameters) into a sequence of machine-level instructions.

The *Processor ISA Library* contains a selection of instructions for the implementation of the macro-operations listed in Table I. It is very important to notice that the *for* statements
implemented in C-like MT are unrolled in the assembler program, in order to have a faster test and to avoid problems due to the presence of branch-prediction strategies in the processor.

Table V and VI show a translation example of two macro-operations for the Nios II and MicroBlaze ASM, respectively. These tables display the first translation step, from macro-operations to Meta-ISAs [16], and the second one from Meta-ISAs to ASM, obtained applying the Target processor ISA description of the target processor.

### IV. Experimental Results

The MarciaTesta tool has been used to generate the test for the data cache memories of two different microprocessors: Microblaze [17] and Nios II [18].

Microblaze is a soft core processor designed for Xilinx FPGAs, with a Harvard memory architecture, a RISC-like instruction set and a data cache with write-through policy. Nios II is a 32-bit RISC embedded-processor designed for the Altera FPGAs with a data cache with write-back policy. The actual boards on which we deployed the automatically generated tests are a Virtex4 ML-403 Embedded Platform [19] and a Nios Development Board Cyclone II Edition [20], respectively.

Both processors have been implemented with 2kB instruction and data cache with four words per cache line, 128kB of on-chip memory. This means that $T = 6$, $I = 9$ and $O = 2$ (see Fig. 2).

Table VII shows some data related to the automatic generation of the assembly programs to implement 11 different March Tests. For each test, the table lists the number of assembly instructions of the test program generated by MarciaTesta for both the data and the directory array test.

One can notice that there is no correlation between the complexity of the original march test and the number of instructions of the test program. This can be easily understood by focusing on the write and read operations. While these are usually considered as atomic in "traditional" March test, i.e., in March tests for "normal" memories, when targeting cache memories they must be implemented resorting to a sequence of instructions.

In addiction, the MicroBlaze test requires less assembly instructions than the Nios II. This is due to the reduced number of operations required to create the cache incoherence during the write operation in the directory array test of the write-through cache.

### V. Conclusion

In this paper we presented MarciaTesta, an EDA tool for the automatic generation of assembly cache test program. Starting...
from a formal definition of March Test, MarciaTesta generates assembly test code for the target processor. The cross-architecture feature of the tool, i.e., its capability of generating assembly code for several architectures, relies on the availability of a set of libraries describing the ISA of the target processors.

The correctness of the tool has been proved by a novel validation and verification approach based on the generation of log files at three different abstraction levels, and namely at the March Test level, at the C implementation level and, eventually at the machine Instruction level. The correctness of the generated code has been checked on several target architectures and in the implementation of several test algorithms on each of them.

Next releases of MarciaTesta will support from the one hand set-associative cache memories and, from the other hand, instruction caches.

ACKNOWLEDGMENT

The authors would like to express their sincere thanks to the whole design team of Ansaldo STS SpA for their helpful hints and guidelines.

REFERENCES