The SimpleScalar Macro Tool (SSIT)

Carsten M. van der Hoeven, B.H.H. (Ben) Juurlink, Demid Borodin
Computer Engineering Laboratory
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology
Mekelweg 4, 2628 CD Delft, The Netherlands
Phone: +31 15 2787362, Fax: +31 15 2784898.
E-mail: C.M.vanderHoeven@TUDelft.nl, {benj,demid}@ce.et.tudelft.nl

Abstract— **SimpleScalar is a simulation tool set that is often used in computer architecture research. Amongst others, it provides a mechanism to synthesize new instructions without having to modify the assembler, by annotating existing instructions in the assembly files. This mechanism, however, is rather error-prone, especially for a novice in SimpleScalar. For this reason two tools have already been developed. These tools are called SimpleScalar Instruction Tool (SSIT) and SimpleScalar Architecture Tool (SSAT). With SSIT, a developer can add functional units and new instructions to SimpleScalar and it allowes the programmer to use readable instructions in assembly to make programming ISA extensions less error prone and easier to do. SSAT enables the developer to extend the number of registers in the SimpleScalar simulator and use them in assembly. This paper presents the SimpleScalar Macro Tool (SSMT), a tool to enable the developer to write ISA extensions in C programming language instead of in assembly. This makes extending the SimpleScalar simulation toolset with new instructions much easier because the developer now can concentrate on writing optimal code without having to bother about register allocation and instruction ordering of the existing ISA. Besides that, experiments conducted on several kernels have showed that SSMT does not introduce a significant performance overhead and in some cases even increases the performance of ISA extensions up to 6% compared to hand written assembly code.**

Keywords— **processor simulator, instruction set architecture, macro programming interface**

I. Introduction

Simulation is often used to evaluate newly designed computer architectures. SimpleScalar [1, 2] is a very popular processor simulator toolset that enables a developer to effectively implement, simulate and evaluate ISA extensions. To ensure easy development, a version of the GNU C compiler has been provided along with the toolset. SimpleScalar has been used to evaluate many new architectural designs such as novel branch predictors [3,4], cache organizations [5,6] , instruction set extensions [7, 8] and fault tolerance schemes [9, 10].

The SimpleScalar Processor Toolset provides a mechanism to extend the Instruction Set Architecture, however this mechanism is quite error prone and difficult to use, especially by a novice at SimpleScalar. This mechanism requires the developer to write annotated assembly instructions where he needs ISA extensions to be executed. In addition to this, the developer also has to do register allocation for these ISA extensions as well since the provided GNU C compiler does not recognise the extensions. Therefore, two other tools have been designed at the Delft University of Technology. These tools are called SimpleScalar Instruction Tool (SSIT) and SimpleScalar Architecture Tool (SSAT) [11, 12].

SSIT is designed to add functional units and new instructions to SimpleScalar and to allow the user to use readable assembly code instead of annotated assembly code, this makes writing optimized assembly much less error prone and easier. SSAT allows the developer to extend the number of existing registers, alias existing registers and define new registers that are not already present in the SimpleScalar processor and use these in assembly.

Although SSIT and SSAT substantially simplify the work of a developer, ISA extensions still need to be written in assembly language where the developer needs to take everything into account, including register allocation and instruction ordering. To aid the developer in his efforts, the SimpleScalar Macro Tool is developed. This tool allows the developer to write ISA extensions directly in C code and leave the register allocation and instruction ordering to the C compiler.

This paper first discusses SSMT in detail in Section II, where Section II-A covers the SSIT configuration file, Section II-B discusses the internals of SSMT and Section II-C shows a small example. Section III covers the results of the experiments that were conducted to test SSMT and evaluate the performance impact of the macro interface. Finally Section IV provides a short summary.

II. SSMT

The SimpleScalar Macro Tool (SSMT) is a tool designed to work within the tool chain of which the SSIT and SSAT tools are already part of. Unlike SSIT and SSAT, which process assembly code to change readable mnemonics into annotated instructions that are recognized by the assembler, SSMT is used before compiling C-code into assembly. SSMT takes in the SSIT configuration file and generates a header file with C macros that can be used throughout any C code that needs to be optimized using the newly designed ISA extensions described in the configuration file. This is depicted in Figure 1. This section first describes the SSIT configuration file followed by the internals of SSMT and a small example

A. SSIT configuration file

In order to run SSIT and SSMT properly, a configuration file needs to be provided. The contents and the format of this file has been defined by the developers of SSIT, but is very suitable to provide SSMT the appropriate information. A SSIT configuration file consists of a number of sections with one of two types: (1) FUNCTIONAL_UNITS sections in which new functional units can be defined or (2) MACHINE.DEF sections that allows a developer to define new instructions. Since the FUNCTIONAL_UNITS sections are not used by SSMT they are not discussed further. The MACHINE.DEF sections are constructed as follows:

```
#define <INSTRUCTION NAME>_IMPL { \
<Functional implementation of the new instruction in C> \
}
DEFINST(<instruction name>, <operands>,
<name of annotated instruction>,
<FU class>, <iflags>,
<odep1>, <odep2>, <idep1>, <idep2>, <idep3>
)
```

where

`<instruction name>` is the name of the new instruction as a string.

`<operands>` is a string that specifies the instruction operand fields (the instruction format).

`<name of annotated instruction>` is the name of an existing SimpleScalar instruction which will be annotated to pass the new instruction through the SimpleScalar assembler.

`<FU class>` is the resource class required by this new instruction.

`<iflags>` are instruction flags, used by SimpleScalar for fast decoding.

`<odep1>, <odep2>` are two output dependency designators. They specify which registers are modified by the instruction and are used to determine data dependencies.

`<idep1>, <idep2>, <idep3>` are input dependency designators. They specify which registers are read by the instruction.

For more information on SSIT please refer to the SSIT manual [12].

B. Internals

Figure 1 depicts how SSMT is used in combination with SSIT and SSAT. The SSIT configuration file is described in Section II-A and in [12]. This section will describe how the SSIT configuration file is used to generate the header file with the macros.

The data that is necessary to construct the macros is taken from each instruction definition in the SSIT configuration file. From this file, the following items are taken from each MACHINE.DEF section: <instruction name>, <operands>, <odep1>, <odep2>, <idep1>, <idep2> and <idep3>. From the information that is obtained by reading these fields, the macros are constructed. The general form of these macros is:

`SSMT_INSTRUCTION_NAME_XXXXX(ARGUMENTS)`

where

`SSMT` is a prefix to ensure that each macro generated with SSMT is unlike a user defined macro.

`INSTRUCTION_NAME` is the name of the instruction in capitals. If an instruction name contains dots (spaces and underscores are not allowed), and underscore should take its place.

`XXXXX` is a sequence of letters indicating the type of the following arguments. Each X can stand for one of the following four possibilities:

- *Nothing* - If the number of arguments is less than 5, some of the letters are omitted and do not appear in the macro.
- *V* - If the argument is supposed to be a variable name, the letter is a 'V'.
- *R* - If the argument is supposed to indicate a register, the letter is an 'R'. The register name should be indicated by a string for correct results. More about this comes later.
- *I* - For the instructions that need an immediate operand, there is the letter 'I' to indicate that the variable should be an immediate value.

`ARGUMENTS` is a list of arguments in the same sequence as they are needed by the instruction to which the macro refers. The type of these arguments should be as determined by the letters preceding the arguments. Compiler errors will occur when the types are not cor-

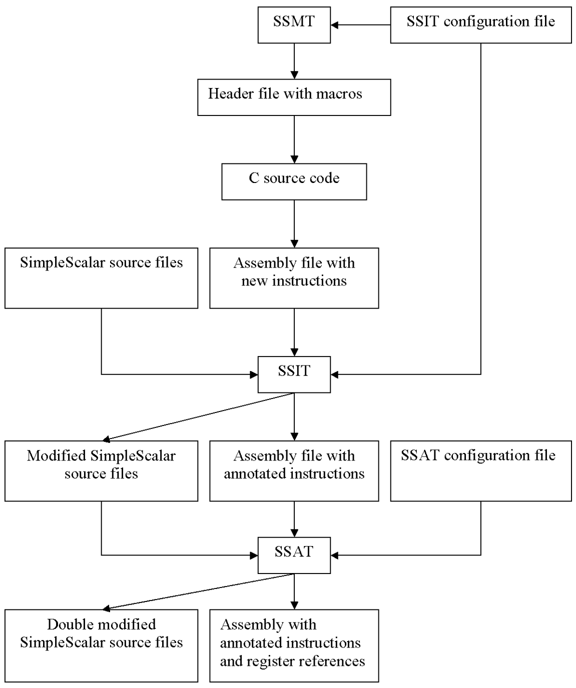
SSMT
SSIT configuration file
Header file with macros
C source code
SimpleScalar source files
Assembly file with new instructions
SSIT
Modified SimpleScalar source files
Assembly file with annotated instructions
SSAT configuration file
SSAT
Double modified SimpleScalar source files
Assembly with annotated instructions and register references

Fig. 1
Entire toolchain around SimpleScalar, including SSIT, SSAT and SSMT

rect, although no type checking is done. The need for differentiation by letters comes from the fact that the macros are regular C macros. Therefore, each macro should be unique and given the number of possibilities, this is the most convenient way to ensure that all the macros are unique.
The distinction between variable names and register names (strings) originates from the possibility to extend the number of registers or define new registers with SSAT. Since the compiler does not recognize these new registers, it is not possible to apply automatic register allocation on those registers. Therefore, a developer needs to do manual register allocation for extended or new registers by the use of register names, but can leave the register allocation to the compiler when using existing registers by using the variable names.
Due to minour differences in the implementation of some instructions, there are two macros that always have the same configuration:

- **Comment macro**
This is a macro that has "COMMENT" for the instruction name and takes only a string as an argument. No identification letters are needed.
- **Load and Store instructions**
Instructions that operate on memory always have the following macros to access them:
`SSMT_NAME_RV(register_name, pointer_to_variable)`
and
```
SSMT_NAME_RVI(register_name, pointer_to_variable,
  immediate_address_offset)
```
Because custom load and store instructions are not written unless there is a need for them (because of new types of registers that are defined with SSAT), these macros take in the name of a register and the address of a certain variable (which can be loaded into the new register). For easy memory offset (which is possible in SimpleScalar), the immediate value is added since the translation from macro to assembly does not do this automatically.
The macros that are described above are regular C macros. This means that they are simply replaced during the pre-processing of the C file. In this case each macro is replaced by a piece of extended in-line assembly containing the name of the instruction and all the variables.

C. Example

This section demonstrates an example of the use of the macros in C-code. The example consists of snippets taken from four different files. The first snippet is taken from the SSIT configuration file:

```
#define AVG_IMPL {                              \
  sword_t result = (GPR(RS)+GPR(RT)) >> 1; \
  SET_GPR(RD, result);              \
}

DEFINST( "avg", "d,s,t", "add",
         avg_FU, F_ICOMP,
         DGPR(RD), DNA, DGPR(RS), DGPR(RT), DNA
       )
```

This piece of code defines a new instruction that calculates the average of two input arguments and writes it to the output register. In the instruction definition, the name of the annotated instruction, "avg" in this case, comes first, then the the register layout followed by the instruction that should be annotated to make the new instruction available in the simulator. Next comes the functional unit followed by an instruction flag. Last come the output and input registers, each of which one is not used (DNA). For more information about this see Section II-A or [12].

Given the SSIT configuration file with the instruction definition as stated above, SSMT generates the following in the header file:

```
SSMT_AVG_VVV(out0,in0,in1) asm("avg %0,%1,%2":"=r" (out0):"r" (in0),"r" (in1))
SSMT_AVG_VVR(out0,in0,in1) asm("avg %0,%1," in1 "":"=r" (out0):"r" (in0))
SSMT_AVG_VRV(out0,in0,in1) asm("avg %0," in0 ",%1":"=r" (out0):"r" (in1))
SSMT_AVG_VRR(out0,in0,in1) asm("avg %0," in0 "," in1 "":"=r" (out0):)
SSMT_AVG_RVV(out0,in0,in1) asm("avg " out0 ",%0,%1"::"r" (in0),"r" (in1))
SSMT_AVG_RVR(out0,in0,in1) asm("avg " out0 ",%0," in1 ""::"r" (in0))
SSMT_AVG_RRV(out0,in0,in1) asm("avg " out0 "," in0 ",%0"::"r" (in1))
SSMT_AVG_RRR(out0,in0,in1) asm("avg " out0 "," in0 "," in1 ""::)
```

Since the new instruction takes in three different arguments and each argument can be either a normal variable that is placed in a certain register or a variable that is placed in a type of register that is either new, extended or aliased, the total number of possibilities is eight.

An example of a C source file in which the generated macro is used is stated below:

```
#include "ssmt.h"

int main(int argc, char ** argv)
{
  short a = 6, b = 4;
  SSMT_AVG_VVV(a, a, b);
  return 0;
}
```

Here, the macro "SSMT_AVG_VVV" takes in variables 'a' and 'b' and stores the result in 'a'. Note that this code does not present its result to the user, however at the end, variable 'a' should contain the value '5'.

The last part of the example is a small piece of the code that is generated by gcc, with optimization level 2 enabled:

```
..
  li    $5, 0x00000006
  li    $2, 0x00000004
#APP
  avg   $5,$5,$2
#NO_APP
..
```

As can be seen in the assembly code, the variables 'a' and 'b' are loaded into a register before execution of the new instruction. The piece of code where the variables are loaded into registers is automatically generated by GCC.

III. Experimental results

To test the impact of the macro tool on the performance of applications using ISA extensions, eight multimedia kernels that were rewritten in assembly to use an adapted form of MMX extensions are now rewritten in C code to use the macros SSMT created. The rewritten C code is then compiled with GCC and ran through SSIT and SSAT to obtain executable code. The rewritten C code versions of the kernels are then compared to their hand written counterpart and the original C code. First the kernels themselves will be discussed, followed by the results of the experiments.

A. Kernels

In order to make proper comparison a total of eight multimedia kernels were changed to use the macros from the header file that was created with SSMT. All kernels are optimized for the use of MMX instructions and thus work on vectors of four or eight bytes at a time. Three of the kernels have actually two forms that are basically the same. One of the two is a small version that works on matrices of eight by eight or eight by sixteen bytes. The other one is a large version that works on vectors and matrices with much larger dimensions to more accurately simulate the result of the use of MMX instructions because relative overhead for example is decreased.

A.1 Add image

This kernel adds two images of the same size and stores the result in one of the input images. The images are matrices of unsigned bytes thus limiting each value between 0 and 255. Saturation is done in the MMX instructions.

The small version of the 'Add image' kernel adds two images of 8 x 8 bytes.

The large version of the 'Add image' kernel adds two images of 704 x 576 bytes. These images are converted from Windows Bitmap files to be read in properly by the initialization part of the kernel.

A.2 DCT

The DCT kernel computes the Discrete Cosine Transformation. In this particular case the algorithm works on a matrix of 704 x 576 bytes and stores the result in a different one.

A.3 IDCT

To calculate the reverse process of the DCT, this kernel calculates the Inverse Discrete Cosine Transformation. Once again this kernel operates on a matrix of bytes that is 704 high and 576 wide. The result is stored in a different matrix than the original one.

A.4 Matrix Transpose

The transpose of a matrix is calculated with this kernel. Like the kernel that adds two images this kernel has two versions:
The smaller version transposes an 8 x 8 matrix, the larger matrix (for the large version) transposes a matrix of 704 x 576.

A.5 Matrix Vector multiplication

Matrix Vector multiplication is something that is used quite often. This kernel also has a version that works on a matrix of 8 x 16 bytes and a vector of 16 bytes, this is the small one. The large version works on a matrix of 704 x 576 bytes and a vector of 576 bytes.

B. Comparison results

This chapter discusses the results of the comparison between hand written code and C-code using macros. The total comparison is between the original C-code, C-code optimized with GCC optimization level 2, hand written assembly, C-code using macros and C-code using macros and optimized with GCC optimization level 2.

As a measurement, the total number of cycles that is needed to execute a certain kernel is taken and compared to the other values. The results of the experiment is shown in Figure 2. From this figure it is clear that the hand written assembly code performs significantly better than the original C code, however, optimizing C code with GCC already proves to give a significant speedup that is in half the cases even faster than the unoptimized C code that uses the macro-interface. Optimizing the macro using C-code with GCC gives performance that is fairly comparable to hand written assembly, in some cases the macro using optimized C-code is even better than the hand written assembly. To make the relation between the performance of the hand written assembly and the optimized C code using the macro interface, both values are normalized to the number of cycles of the hand written version and shown in Figure 3. This figure shows that, with the exception of one case, the optimized C code has performance that rivals that of the hand written code. The differences in performance can be partially explained by the following:

- **FOR loops**

FOR loops are statements that are widely used by programmers. The way a FOR loop is translated into assembly is in general dependent on the way it is written in C. An initialization or loop condition with a certain variable for example will give a different result than the same thing with values that are known at compile time. GCC makes these kind of decisions based on what is known at compile time, without regards to the context of the loop (something that the compiler can not recognize because it lacks the intelligence to do so). A human programmer however, can take the context into account and therefore write either a more efficient FOR loop or (unconsciously) a less efficient one.

- **registers**

When it comes to registers, the compiler is mostly better than human programmers. Where GCC has no problem at all taking all available registers into account, a human programmer has far more difficulties in doing so. For that reason a human programmer will most likely give up his efforts before they begin and take a decrease in performance for granted to be able to write code that is more understandable (and thus easier to write), where GCC writes code that is less readable but more efficient when it comes to registers.

IV. Conclusion

This paper has presented the SimpleScalar Macro Tool. A tool that allows a developer to use new instructions that are written for the SimpleScalar simulator in a high level programming language by means of a macro interface. The macro interface simplifies the use of ISA extensions when programming applications that use them. Although the programmer is not completely relieved from programming new applications in assembly language, SSMT does make the job easier by removing the burden of register allocation and all other non-ISA extension related programming from the programmers mind. Besides that,

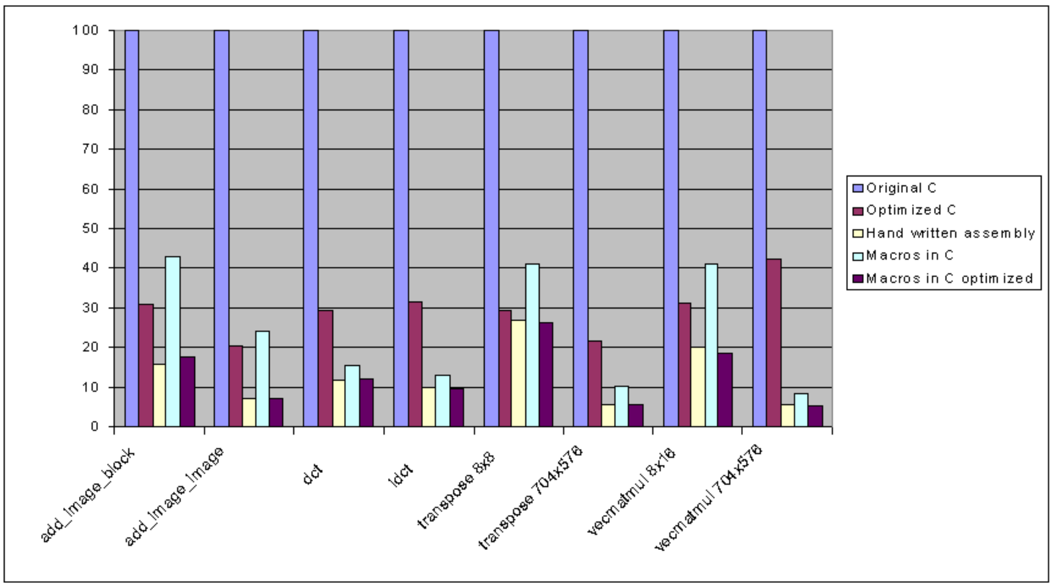

Fig. 2
Number of execution cycles that is needed to perform a kernel, normalized to the execution time of the unoptimized original C code

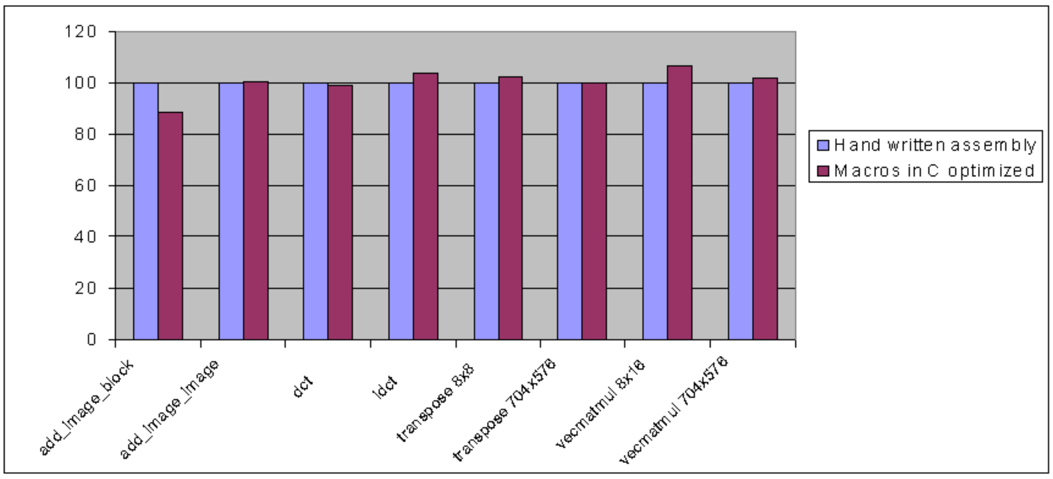

Fig. 3
Number of execution cycles that is needed to execute a kernel, normalized to the number of cycles needed by the hand written assembly

SSMT usually does not introduce a significant performance decrease that is often associated with the use of a macro(-like) interface. In some cases the macro using C code even outperforms the hand-written code.

References

[1] Todd Austin, Eric Larson, and Dan Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *Computer*, 35(2):59–67, 2002.

[2] Doug Burger and Todd M. Austin. The SimpleScalar Tool Set, Version 2.0. *SIGARCH Comput. Archit. News*, 25(3):13–25, 1997.

[3] Daniel A. Jimenez. Piecewise Linear Branch Prediction. In *Proc. 32nd Annual Int. Symp. on Computer Architecture (ISCA-05)*, pages 382–393, Washington, DC, USA, 2005. IEEE Computer Society.

[4] Renju Thomas, Manoj Franklin, Chris Wilkerson, and Jared Stark. Improving Branch Prediction by Dynamic Dataflow-based Identification of Correlated Branches From a Large Global History. In *Proc. 30th Annual Int. Symp. on Computer Architecture (ISCA-05)*, pages 314–323, New York, NY, USA, 2003. ACM Press.

[5] Serkan Ozdemir, Debjit Sinha, Gokhan Memik, Jonathan Adams, and Hai Zhou. Yield-aware cache architectures. In *Proc. 39th Annual IEEE/ACM Int. Symp. on Microarchitecture (MICRO-39)*, pages 15–25, Washington, DC, USA, 2006. IEEE Computer Society.

[6] Chuanjun Zhang. Balanced Cache: Reducing Conflict Misses of Direct-Mapped Caches. In *Proc. 33rd Annual Int. Symp. on Computer Architecture (ISCA-06)*, pages 155–166, Washington, DC, USA, 2006. IEEE Computer Society.

[7] D. Cheresiz, B.H.H. Juurlink, S. Vassiliadis, and H.A.G. Wijshoff. The CSI Multimedia Architecture. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, pages 1–13, January 2005.

[8] Asadollah Shahbahrami, Ben Juurlink, Demid Borodin, and Stamatis Vassiliadis. Avoiding Conversion and Rearrangement Overhead in SIMD Architectures. *International Journal of Parallel Programming*, pages 237–260, June 2006.

[9] Eric Rotenberg. AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors. In *FTCS-29*, pages 84–91, Madison, Wisconsin, USA, Jun 1999. IEEE Computer Society Press.

[10] D. Borodin, B.H.H. Juurlink, and S. Vassiliadis. Instruction-Level Fault Tolerance Configurability. In *IC-SAMOS VII: Int. Conf. on Embedded Computer Systems: Architectures, Modeling, and Simulation*.

[11] B.H.H. (Ben) Juurlink, Demid Borodin, Roel J. Meeuws, Gerard Th. Aalbers, and Hugo Leisink. The SimpleScalar Instruction Tool (SSIT) and the SimpleScalar Architecture Tool (SSAT). unpublished manuscript, available at http://ce.et.tudelft.nl/ demid/SSIAT/.

[12] SSIT, SSAT and SSIAT webpage. `http://ce.et.tudelft.nl/~demid/SSIAT/`.