C++ FSM frameworks comparison

Krzysztof Jusiak

June 30, 2013

Abstract

Document presents comparison between C++ FSM frameworks:

- Boost Meta State Machine (msm)
- Boost State Chart (statechart)
- Quick Finite State Machine (QFsm)

Contents

Chapter 1

Introduction

1.1 Finite state machine

Unified Modeling Language is a standardized general-purpose modeling language in the field of object-oriented software engineering. The standard is managed, and was created by, the Object Management Group.
http://www.omg.org/spec/UML

State diagram is a type of diagram used in computer science and related fields to describe the behavior of systems. State diagrams require that the system described is composed of a finite number of states; sometimes, this is indeed the case, while at other times this is a reasonable abstraction. There are many forms of state diagrams, which differ slightly and have different semantics.
http://en.wikipedia.org/wiki/State_diagram

Finite state machine is a mathematical abstraction sometimes used to design digital logic or computer programs. It is a behavior model composed of a finite number of states, transitions between those states, and actions, similar to a flow graph in which one can inspect the way logic runs when certain conditions are met. It has finite internal memory, an input feature that reads symbols in a sequence, one at a time without going backward; and an output feature, which may be in the form of a user interface, once the model is implemented. The operation of an FSM begins from one of the states (called a start state), goes through transitions depending on input to different states and can end in any of those available, however only a certain set of states mark a successful flow of operation (called accept states).
http://en.wikipedia.org/wiki/Finite-state_machine

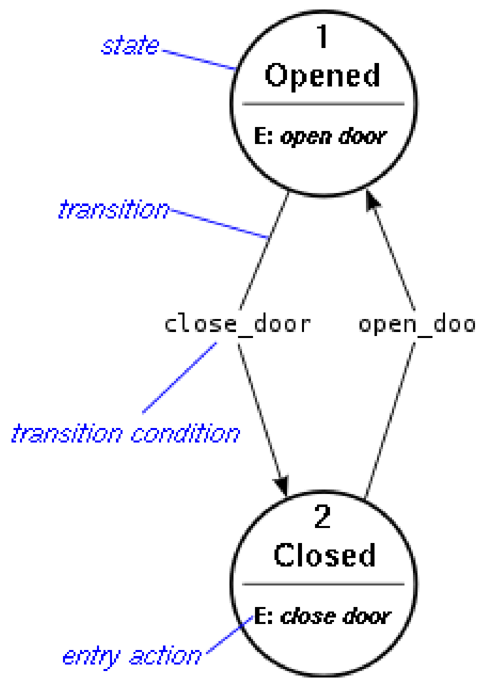


Figure 1.1: FSM

The Domain Abstraction ion of finite state machines consists of three simple elements.

- **States**
  An FSM must always be in one of several well-defined states. For example, the states of a simple CD player might be called Open, Empty, Stopped (with a CD in the drawer), Paused, and Playing. The only persistent data associated with a pure FSM is encoded in its state, though FSMs are seldom used alone in any system. For example, the parsers generated by YACC are built around a stack of state machines; the state of the whole system includes that of the stack and of each FSM in the stack.

- **Events**
  State changes are triggered by events. For example in CD player example, most events would correspond to button presses on its front panel: play, stop, pause, and open/close (the button that opens and closes the drawer). Events aren't necessarily "pushed" into a state machine from the outside, though. For example, in YACC parsers, each event represents a different token, and is "pulled" from the input stream by the parsing process. In some systems, events contain associated data. For instance, an identifier token in a C++ parser might carry the text of the identifier, while an integer-literal token might carry the value of the integer.

- **Transitions**
  Each state can have any number of transitions to other states. Each transition is labeled with an event. To process an event, the FSM follows the transition that starts from the current state and is marked with that event. For example, a CD player has a transition from Playing to Stopped labeled with the stop event. Usually, transitions also have some associated action, such as stop playback in the case of our CD player. In the case of YACC, following transitions means manipulating the stack of FMSs and/or executing the user's semantic actions.

State transition table is a table showing what state (or states in the case of a nondeterministic finite automaton) a finite semiautomaton or finite state machine will move to, based on the current state and other inputs. A state table is essentially a truth table in which some of the inputs are the current state, and the outputs include the next state, along with other outputs.
http://en.wikipedia.org/wiki/State_transition_table

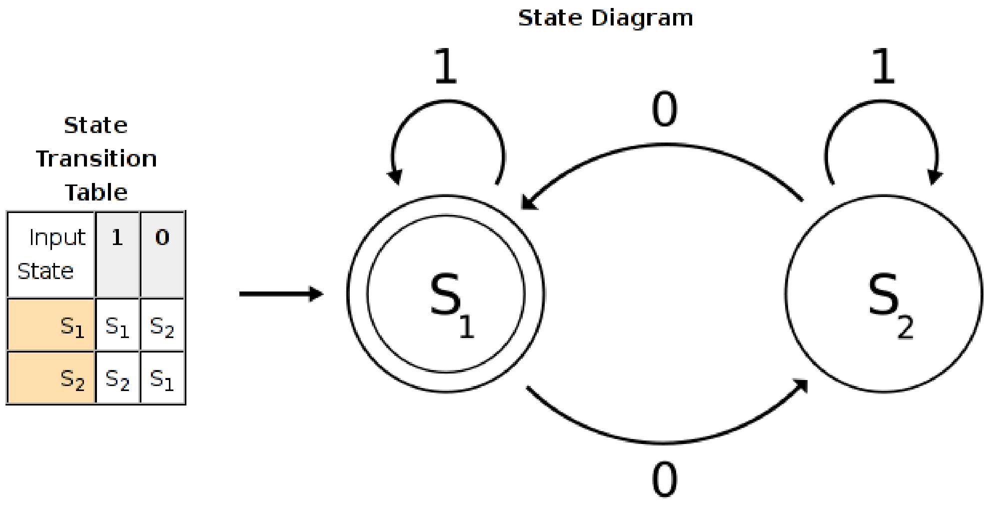


Figure 1.2: State transition table

Virtual finite state machine is a finite state machine (FSM) defined in a virtual environment. The VFSM concept provides a software specification method to describe the behaviour of a control system using assigned names of input control properties and of output actions. The behaviour specification is built by a state table which describes all details of a single state of the VFSM.
http://en.wikipedia.org/wiki/VFSM

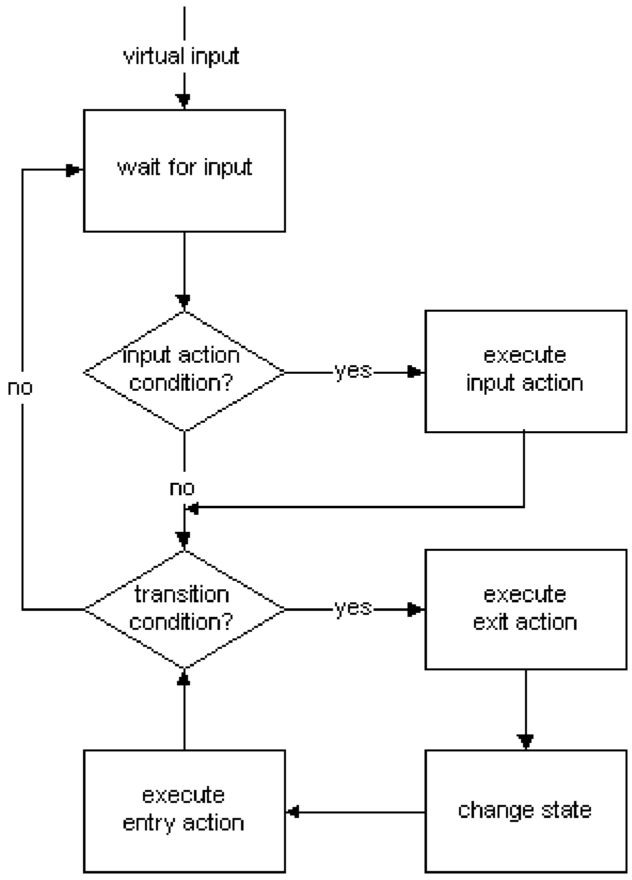


Figure 1.3: VFSM

State table defines all details of the behaviour of a state of a VFSM. It consists of three columns: in the first column state names are used, in the second the virtual conditions built out of input names using the positive logic algebra are placed and in the third column the output names appear.

State Name	Condition(s)	Actions(s)
Current state	Entry action	Output name(s)
	Exit action	Output name(s)
	Virtual condition	Output name(s)
	...	...
Next state name	Virtual condition	Output name(s)
Next state name	Virtual condition	Output name(s)
...	...	...

Figure 1.4: State table

Read the table as following: the first two lines define the entry and exit actions of the current state. The following lines which do not provide the next state represent the input actions. Finally the lines providing the next state represent the state transition conditions and transition actions. All fields are optional. A pure combinatorial VFSM is possible in case only where input actions are used, but no state transitions are defined. The transition action can be replaced by the proper use of other actions.

1.2 C++ FSM Frameworks

1.2.1 FSM framework implementation

Some basic concepts how FSM might be realized due to:

- Concept
  - Declarative - transitions are declared at the beginning and can't be changed
  - Imperative - transitions could have happens in custom handling on event
- Approach
  - Compile time - transitions declared during compile time
  - Run time - transitions declared during run time
- Data
  - FSM driven - data in FSM
  - State driven - data in states
  - Action driven - data in actions

1.2.2 State machine example

To visualize differences between frameworks small example was introduced and implemented in all frameworks. State machine *Player* contains 2 states *Open and Close*. Event *OpenClose* trigger transition to the next state (alternately *Open and Close*).

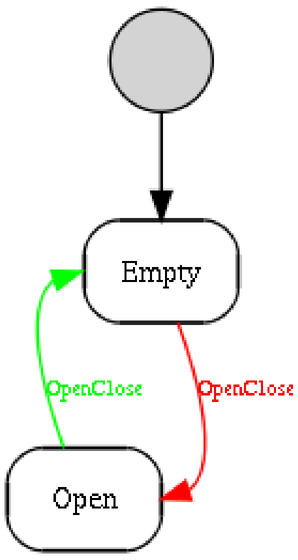


Figure 1.5: Simple state machine

1. **Boost Meta State Machine** (MSM)
   http://www.boost.org/doc/libs/1_47_0/libs/msm/doc/HTML/index.html
   - **Concept** Declarative
   - **Approach** Compile time
   - **Data** State driven, FSM driven

frameworks/msm.hpp

```
class Player_ : public state_machine_def<Player_>
{
    struct Empty : public state<> { };
    struct Open : public state<> { };

public:
    typedef Empty initial_state;

    struct transition_table : boost::mpl::vector
    <
        Row <Empty, OpenClose, Open,  none, none >,
        Row <Open,  OpenClose, Empty, none, none >
    >
    { };
};

typedef boost::msm::back::state_machine<Player_> Player;

Player l_fsm;
```

2. **Boost State Chart** (StateChart)
http://www.boost.org/doc/libs/1_47_0/libs/statechart/doc/index.html
   - **Concept** Declarative, Imperative
   - **Approach** Compile time, Run time
   - **Data** State driven

frameworks/statechart.hpp

```
struct Player : state_machine<Player, Empty> { };

struct Empty : simple_state<Empty, Player>
{
    typedef boost::mpl::vector
    <
        transition<OpenClose, Open>
    >
    reactions;
};

struct Open : simple_state<Open, Player>
{
    typedef boost::mpl::vector
    <
        transition<OpenClose, Empty>
    >
    reactions;
};

Player l_fsm;
```

3. **Quick Finite State Machine** (QFsm)
   - **Concept** Declarative
   - **Approach** Compile time
   - **Data** Action driven

frameworks/qfsm.hpp

```
//Default front end
class Player : public QFsm::Front::Fsm<Player>
{
    struct Empty { };
    struct Open { };

public:
    typedef Empty InitialState;

    typedef boost::mpl::vector
    <
        Transition < Empty, OpenClose, Open  >,
        Transition < Open,  OpenClose, Empty >
    >
    TransitionTable;
};

QFsm::Front::Default::Fsm<Player> l_fsm;

//Default front end in place
QFsm::Front::Fsm
<
    boost::mpl::vector
    <
        Transition < Empty, OpenClose, Open  >,
        Transition < Open,  OpenClose, Empty >
    >,
    Empty
>
l_fsm;

//Fusion front end
BOOST_AUTO(l_transitionTable,
    (
        QFsm::Fusion::TransitionTable<Empty>()
            .transition(Empty(), Event<OpenClose>(), Open())
            .transition(Open(), Event<OpenClose>(), Empty())
    )
);

QFsm::Front::Fusion::Fsm<BOOST_TYPEOF(l_transitionTable)> l_fsm(l_transitionTable);
```

Chapter 2

FSM Comparison

2.1 FURPS

FSM Framework Design Goals

- **Interoperability**
  State machines are typically just an abstraction for describing the logic of a system targeted at some problem domain(s) other than FSM construction. We'd like to be able to use libraries built for those domains in the implementation of our FSMs, so we want to be sure we can comfortably interoperate with other DSELs.

- **Declarativeness**
  State machine authors should have the experience of describing the structure of FSMs rather than implementing their logic. Ideally, building a new state machine should involve little more than transcribing its stat transition table into a C++ program. As framework providers, we should be able to seamlessly change the implementation of a state machine's logic without affecting the author's description.

- **Expressiveness**
  It should be easy both to represent and to recognize the domain abstraction in a program. In our case, an state transitoin table in code should look very much as it does when we design a state machine on paper.

- **Efficiency**
  A simple FSM should ideally compile down to extremely tight code that can be optimized into something appropriate even for a tiny embedded system. Perhaps more importantly, concerns about the efficiency of our framework should never give programmers an excuse for using ad hoc logic where the sound abstraction of a finite state machine might otherwise apply.

- **Static Type Safety**
  It's important to catch as many problems as possible at compile time. A typical weakness of many traditional FSM designs is that they do most of their checking at runtime. In particular, there should be no need for unsafe downcasts to access the different datatypes contained by various events.

- **Maintainability**
  Simple changes to the state machine design should result in only simple changes to its implementation. This may seem like an obvious goal, but it's nontrivial to attainexperts have tried and failed to achieve it. For example, when using the State design pattern, a single change such as adding a transition can lead to refactoring multiple classes.

- **Scalability**
  FSMs can grow to be really complex, incorporating such features as per-state entry and exit actions, conditional transition guards, default and triggerless transitions and even sub-states. If the framework doesn't support these features today, it should be reasonably extensible to do so tomorrow.

FURPS is an acronym representing a model for classifying software quality attributes (functional and non-functional requirements):

- **F**unctionality - Feature set, Capabilities, Generality, Security
- **U**sability - Human factors, Aesthetics, Consistency, Documentation
- **R**eliability - Frequency/severity of failure, Recoverability, Predictability, Accuracy, Mean time to failure
- **P**erformance - Speed, Efficiency, Resource consumption, Throughput, Response time
- **S**upportability - Testability, Extensibility, Adaptability, Maintainability, Compatibility, Configurability, Serviceability, Installability, Localizability, Portability

2.1.1 FURPS and FSM Frameworks

- **Functionality**
  a) UML features / extensions
    - State machine
      * Nested FSMs
      * Orthogonal regions
      * Shallow history
      * Deep history
    - States
      * Normal state
      * Initial state
      * Final state
      * Super state
      * Pseudo state
    - Events
      * Call event
      * Internal event
    - Reactions
      * Guard
      * Transition
      * Deferral
      * Completion/Anonymous transition
      * Internal transition
      * Self transition
      * Conflicting transition (overlapping guards, internal transition in sub fsm and other state)
    - Actions
      * Entry action
      * Exit action
      * Transition action

  b) Extra features
    - Logging (frameworks supports logging events)
- **Usability**
  a) User interface
    - Simple
    - Intuitive
    - Not redundant
- **Repeatability**
  a) Exception safety
- **Performance**
  a) Event processing (dispatching) time
  b) Binary size (debug, release)
  c) Runtime memory consumption (valgrind)
  d) Compilation time
- **Supportability**
  a) Testability (dependency injection)
- **Others**
  a) UML compatibility
  b) UML state diagram generation (how easy is to generate state diagram / scripts existence)

Table 2.1: FURPS FSM comparison

FURPS		MSM	StateChart	QFsm
State machine	Nested FSMs	✓	✓	✓
	Orthogonal regions	✓	✓	✓
	Shallow history	✓	✓	✓
	Deep history	✓	✓	✓
States	Normal state	✓	✓	✓
	Initial state	✓	✓	✓
	Final state	✓	✓	✓
	Super state	-	-	-
	Pseudo state	✓	-	✓
Events	Call event	✓	✓	✓
	Internal event	✓	✓	✓
Reactions	Guard	✓	✓	✓
	Transition	✓	✓	✓
	Deferral	✓	✓	✓
	Completion transition	✓	✓	✓
	Internal transition	✓	✓	✓
	Self transition	✓	-	✓
	Conflicting transition	✓	-	✓
Actions	Entry action	✓	✓	✓
	Exit action	✓	✓	✓
	Transition action	✓	✓	✓
Extra features	Logging	-	-	✓
User interface	Simple	✓	✓	✓
	Intuitive	✓	-	✓
	Not redundant	✓	-	✓
Repeatability	Exception safety	✓	✓	✓
Supportability Others	Testability	✓	-	✓
	UML compatibility	✓	✓	✓
	UML state diagram generation	simple	-	✓

Table 2.2: FSM Framework Design Goals

FSM Framework Design Goal	MSM	StateChart	QFsm
Interoperability	✓	✓	✓
Declarativeness	✓	not all	✓
Expressiveness	✓	✓	✓
Efficiency	✓	-	✓
Static Type Safety	✓	-	✓
Maintainability	✓	✓	✓
Scalability	✓	✓	✓

2.2 Tests

2.2.1 Test transitions efficiency (test_transitions)

tests/test_transitions

```
IN:
    N = e1 event call times
    events { e1 }
    states { S1, S2 }
    fsm
    {
        {S1, e1, S2}
        {S2, e1, S1}
    }
OUT:
    fsm state
    {
        0. -  { S1 }
        1. e1 { S2 }
        2. e1 { S1 }
        ...
        N. e1 { Sx }
    }
```

2.2.2 Test complex state machine (test_complex)

tests/test_complex

```
IN:
    N = event call times (in order e1, e2, e3, e4, e5, e6, e1, e2, ...)
    events { e1, e2, e3, e4, e5, e6 }
    states { S1, S2, S3, S4, S5, S6 }
    fsm
    {
         {S1, e1, S1}
         {S1, e2, S2}
         {S1, e3, S3}
         {S1, e4, S4}
         {S1, e5, S5}
         {S1, e6, S6}
         {S2, e1, S1}
         {S2, e2, S2}
         {S2, e3, S3}
         {S2, e4, S4}
         {S2, e5, S5}
         {S2, e6, S6}
         {S3, e1, S1}
         {S3, e2, S2}
         {S3, e3, S3}
         {S3, e4, S4}
         {S3, e5, S5}
         {S3, e6, S6}
         {S4, e1, S1}
         {S4, e2, S2}
         {S4, e3, S3}
         {S4, e4, S4}
         {S4, e5, S5}
         {S4, e6, S6}
         {S5, e1, S1}
         {S5, e2, S2}
         {S5, e3, S3}
         {S5, e4, S4}
         {S5, e5, S5}
         {S5, e6, S6}
         {S6, e1, S1}
         {S6, e2, S2}
         {S6, e3, S3}
         {S6, e4, S4}
         {S6, e5, S5}
         {S6, e6, S6}
    }
OUT:
    fsm state
    {
        0. -  { S1 }
        1. e1 { S2 }
        2. e2 { S2 }
        3. e3 { S3 }
        4. e4 { S4 }
        4. e5 { S5 }
        4. e6 { S6 }
        5. e1 { S1 }
        ...
        N. ex { Sx }
    }
```

2.2.3 Results from "server" [df6407d], generated Sun Sep 25 23:45:00 CEST 2011

```
Test aspects:

    compilation:
        compilation time measured by 'time' call
        only 'real' time is taken into account
        result is in seconds

    size:
        size of the binary measured by 'ls -k' call
        result is in kilobytes

    strip-size:
        size of the binary measured by 'ls -k' call after 'strip' call
        result is in kilobytes

    execution:
        execution time measured by 'time' call
        only 'real' time is taken into account
        result is in seconds

    valgrind:
        test is executed with valgrind call
        result is as A/D (S), where
        A - allocations
        D - deallocations
        S - global allocated size in bytes

    test name:
        test_NAME[_NUMBER], where NAME is test case name and NUMBER is count of event calls during the test
```

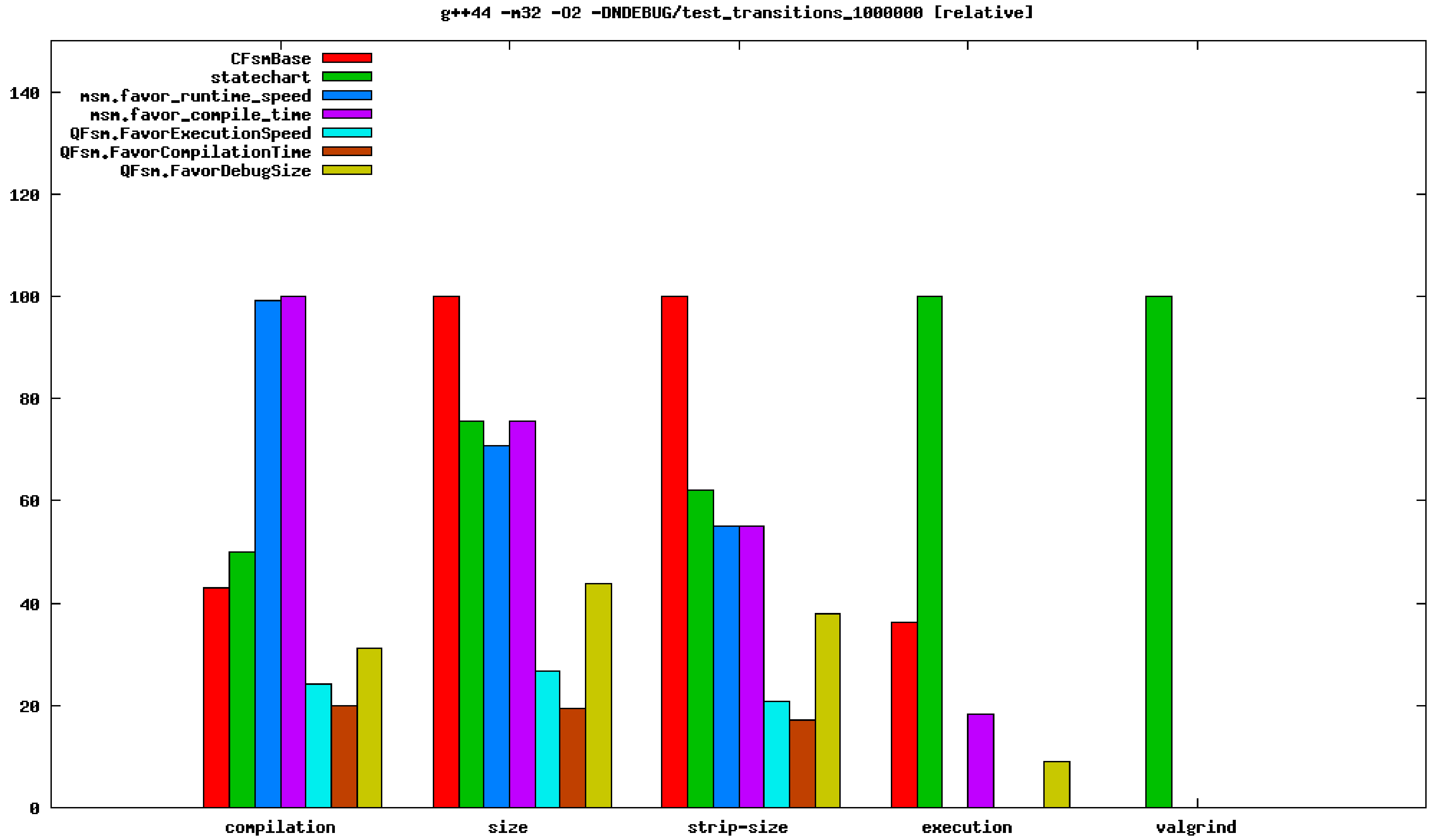
```
Environment statistics:

    generated: Sun Sep 25 23:45:00 CEST 2011
    code revision: df6407d
    hostname: "server"
    operating system:  GNU/Linux
    processor: x86_64
    free memory: 1748Mb
    load average: 1.30 1.39 1.28 1/626 16681
```

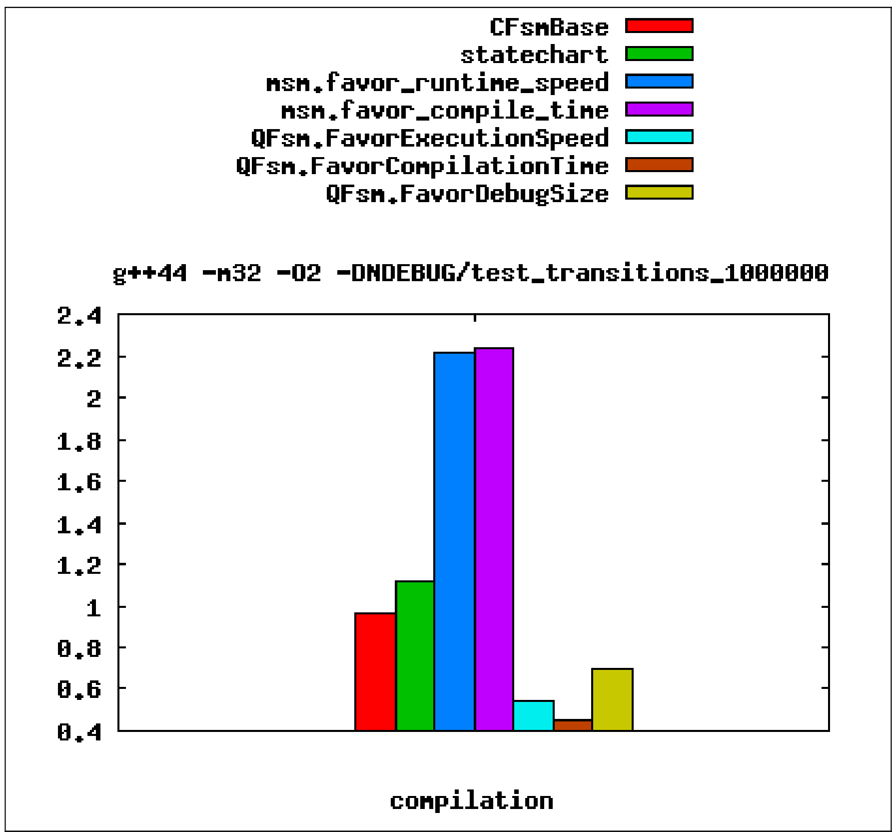
```
All tests summary:

    real: 593.18s (9:53.18)
    user: 579.98s
    sys: 10.70s
    cpu: 99%
    average memory usage: 0K
    maximum resident set size: 2866352K
    number of times the process was swapped out of main memory: 0
    number of file system input: 120
    number of file system outputs: 466288
```

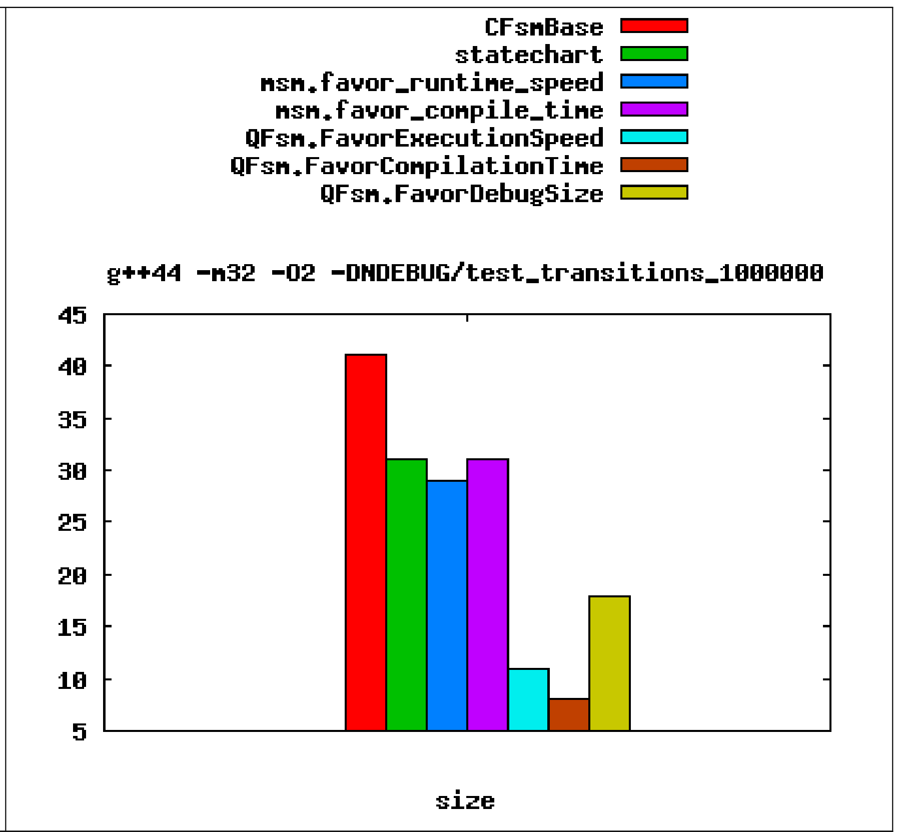
```
Results are presented by using table and two types of charts:

    table: contains results for each tested aspect and framework
    first type of chart: presents relative (0-100%) differents between individual framework and aspect
    second type of chart: presents each aspect individually using exact values returned during the test
```

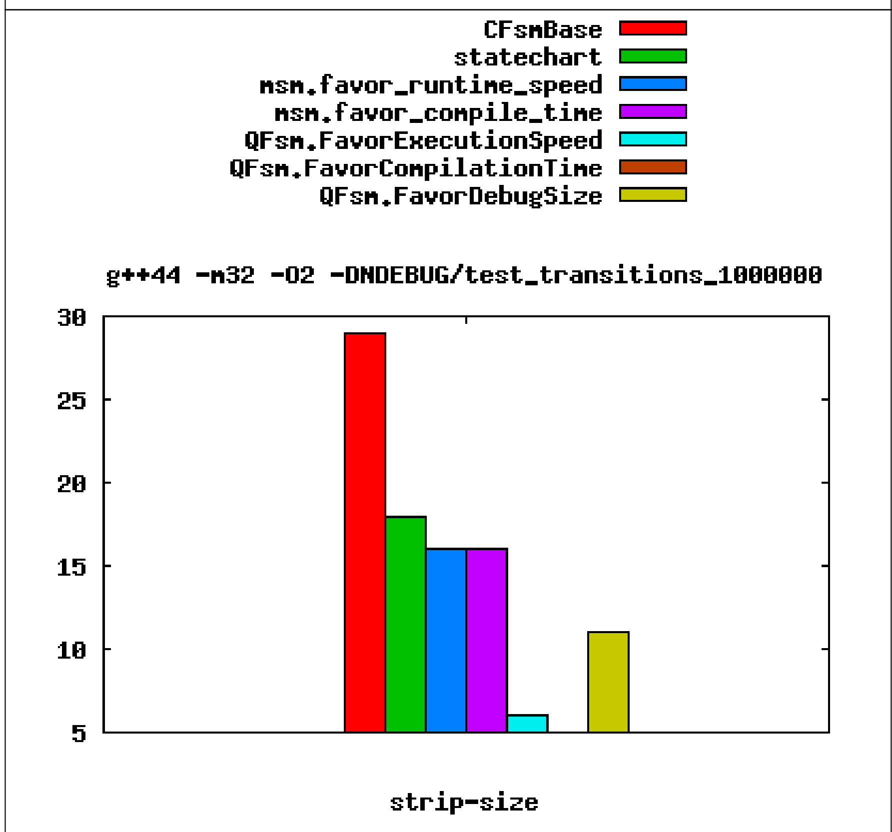
Table 2.3: "server" [54c084f], g++44 -m32 -O2 -DNDEBUG/test transitions 1000000

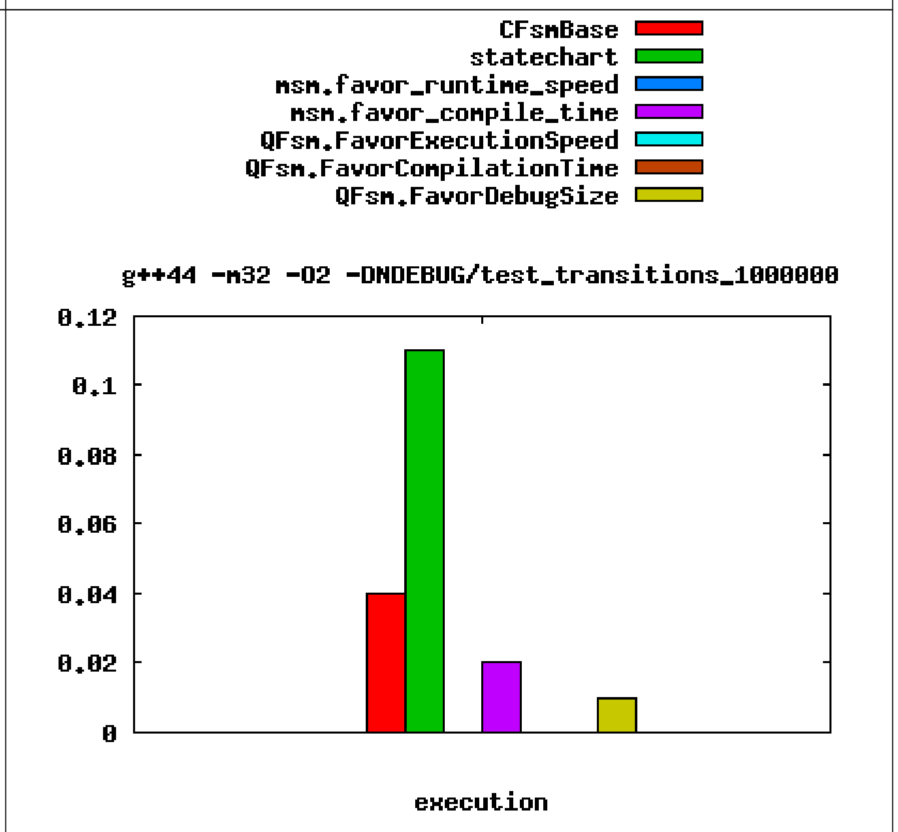
	CFsmBase	StateChart	MSM.favor_runtime_speed	MSM.favor_compile_time	QFsm.FavorExecutionSpeed	QFsm.FavorCompilationTime	QFsm.FavorDebugSize
compilation	0.96s	1.12s	2.22s	2.24s	0.54s	0.45s	0.70s
size	41K	31K	29K	31K	11K	8K	18K
strip-size	29K	18K	16K	16K	6K	5K	11K
execution	0.04s	0.11s	0.00s	0.02s	0.00s	0.00s	0.01s
valgrind	9/9 (138b)	1,000,004/1,000,004 (24,000,064b)	4/4 (561b)	10/10 (2,673b)	2/2 (17b)	2/2 (17b)	16/16 (241b)




CFsmBase
statechart
msm.favor_runtime_speed
msm.favor_compile_time
QFsm.FavorExecutionSpeed
QFsm.FavorCompilationTime
QFsm.FavorDebugSize
g++44 -m32 -O2 -DNDEBUG/test_transitions_1000000
2.4
2.2
2
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
compilation


CFsmBase
statechart
msm.favor_runtime_speed
msm.favor_compile_time
QFsm.FavorExecutionSpeed
QFsm.FavorCompilationTime
QFsm.FavorDebugSize
g++44 -m32 -O2 -DNDEBUG/test_transitions_1000000
45
40
35
30
25
20
15
10
5
size


CFsmBase
statechart
msm.favor_runtime_speed
msm.favor_compile_time
QFsm.FavorExecutionSpeed
QFsm.FavorCompilationTime
QFsm.FavorDebugSize
g++44 -m32 -O2 -DNDEBUG/test_transitions_1000000
30
25
20
15
10
5
strip-size


CFsmBase
statechart
msm.favor_runtime_speed
msm.favor_compile_time
QFsm.FavorExecutionSpeed
QFsm.FavorCompilationTime
QFsm.FavorDebugSize
g++44 -m32 -O2 -DNDEBUG/test_transitions_1000000
0.12
0.1
0.08
0.06
0.04
0.02
0
execution

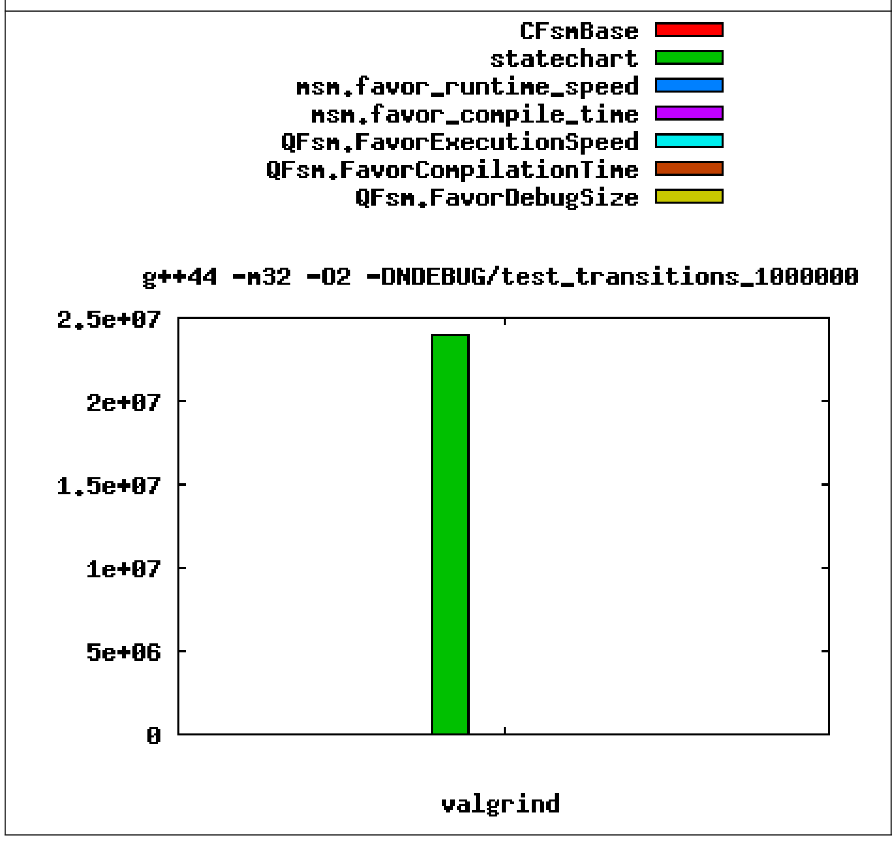
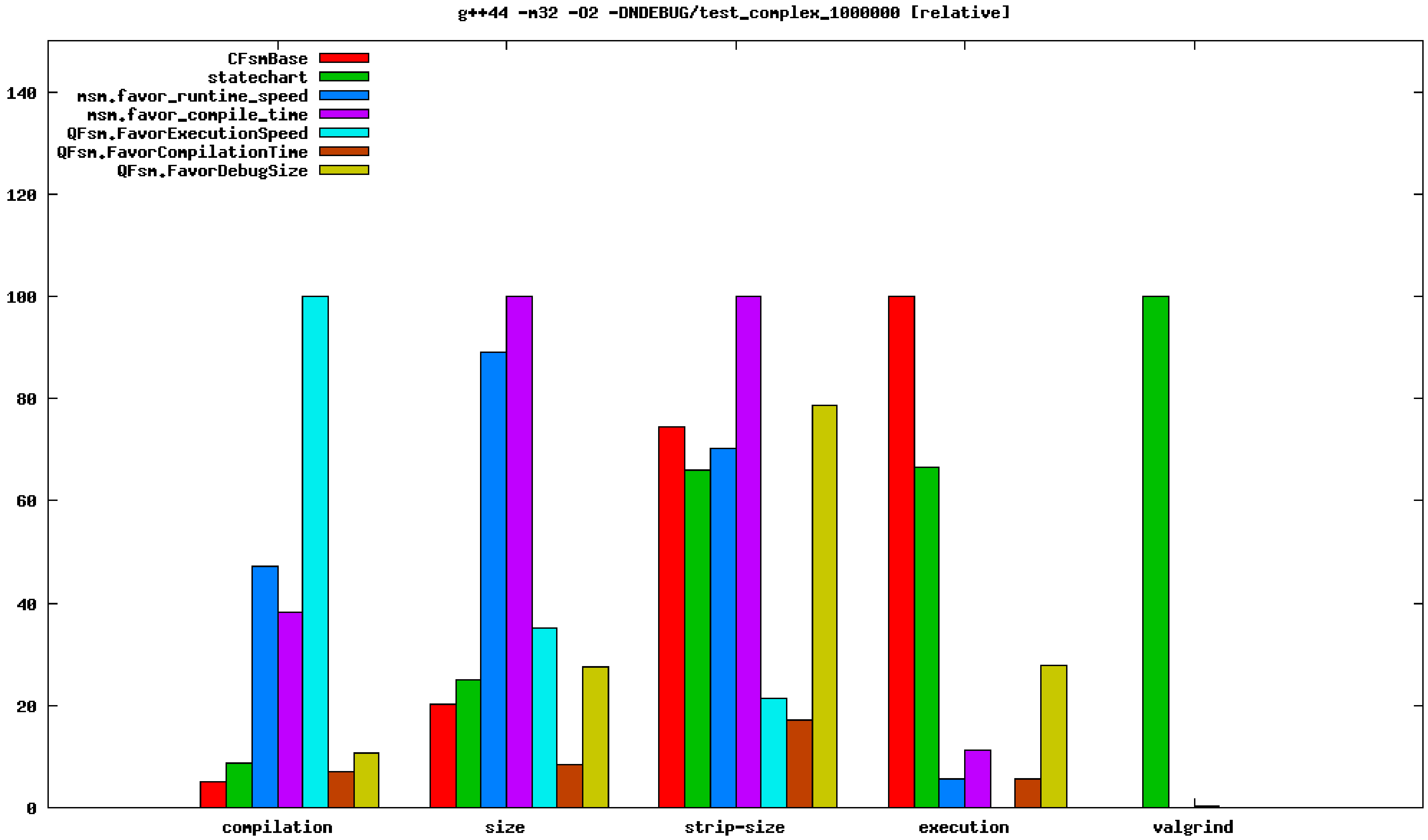
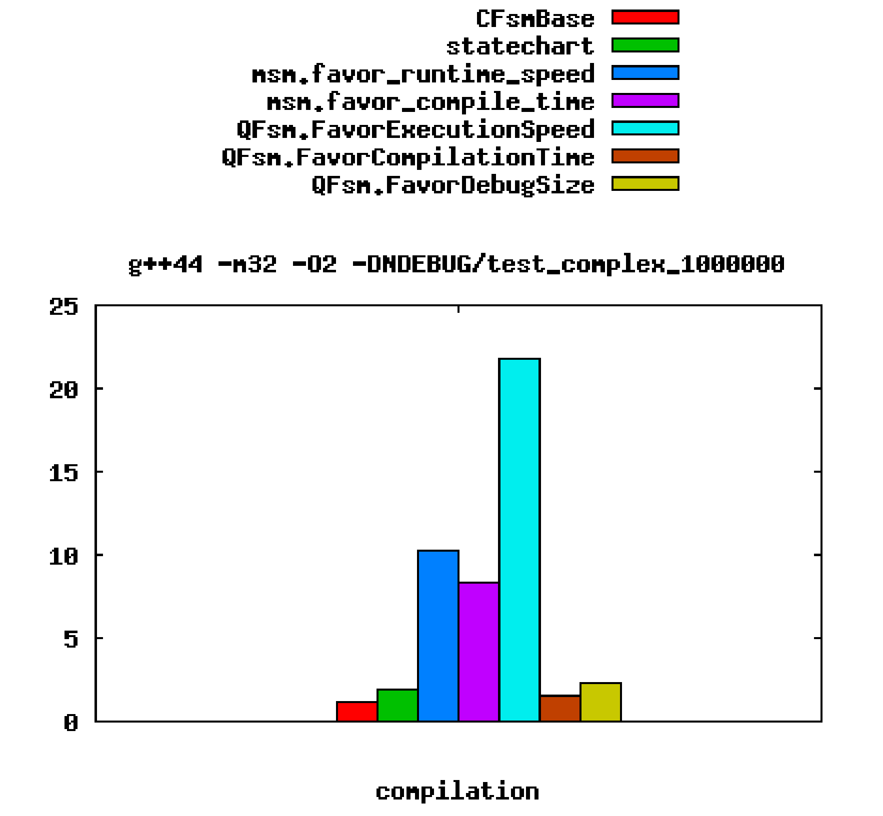
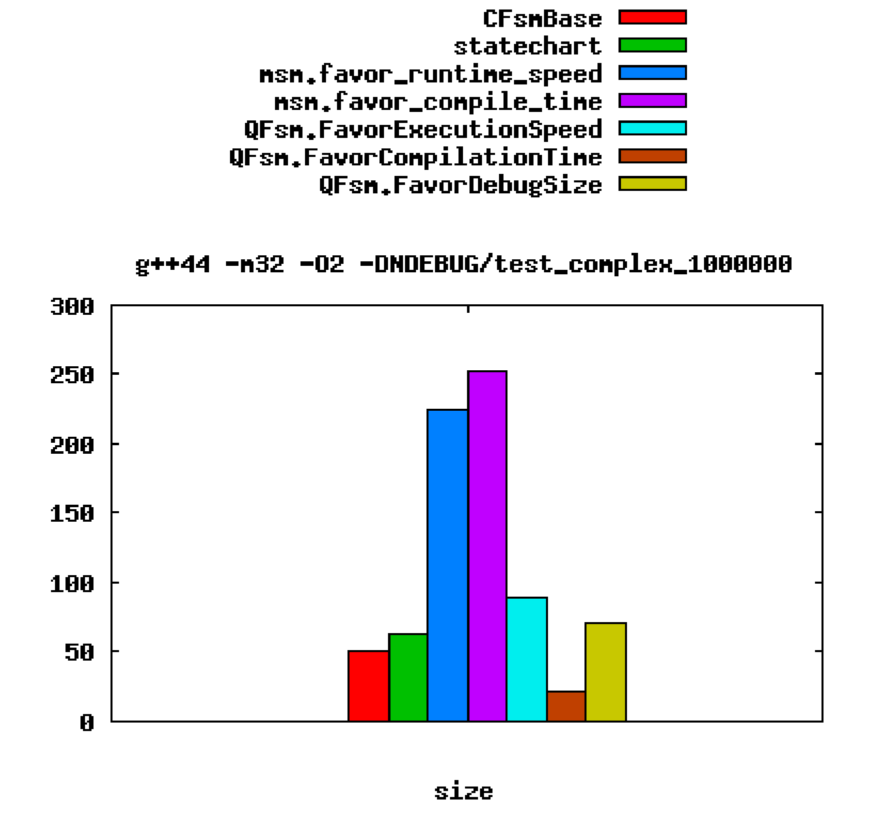

CFsmBase
statechart
msm.favor_runtime_speed
msm.favor_compile_time
QFsm.FavorExecutionSpeed
QFsm.FavorCompilationTime
QFsm.FavorDebugSize
g++44 -m32 -O2 -DNDEBUG/test_transitions_1000000
2.5e+07
2e+07
1.5e+07
1e+07
5e+06
0
valgrind

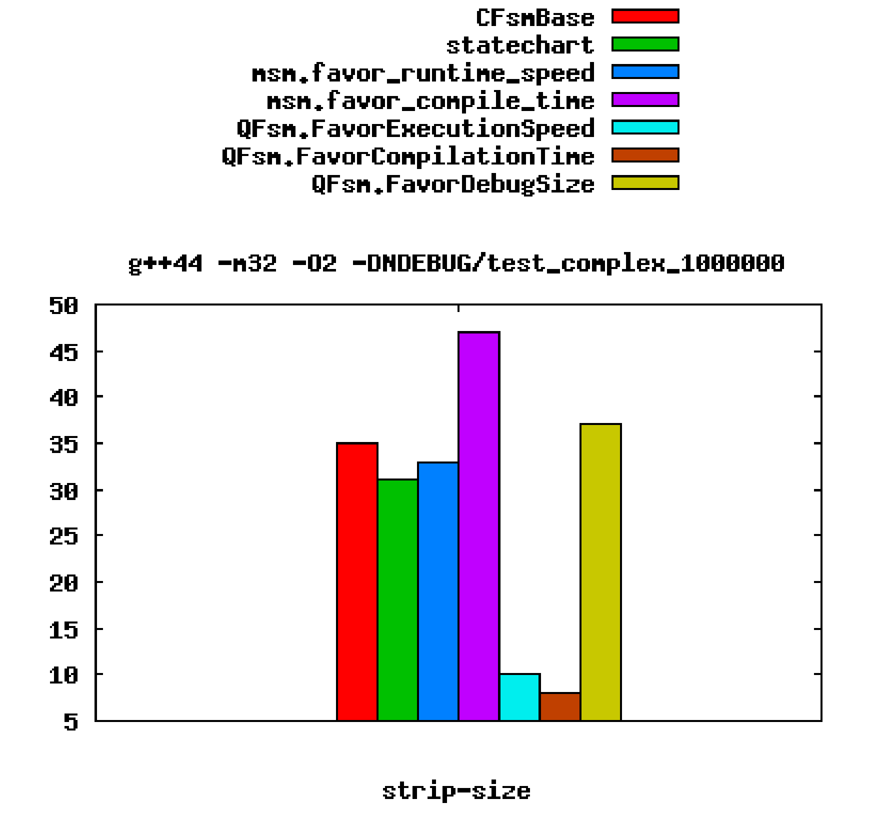
Table 2.6: "server" [54c084f], g++44 -m32 -O2 -DNDEBUG/test complex 1000000

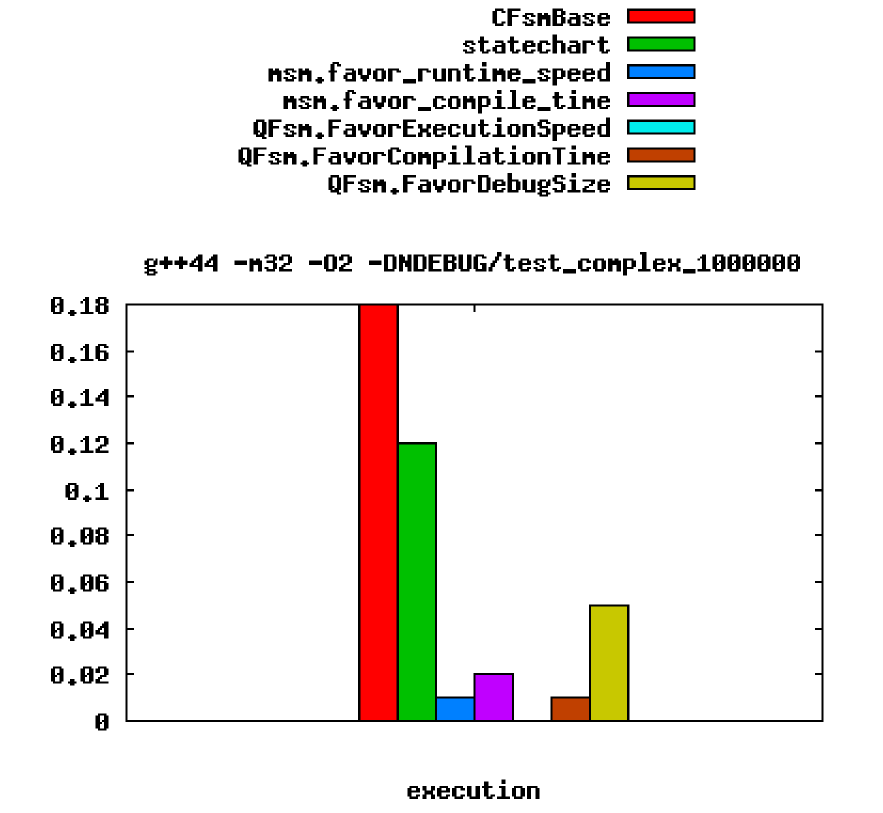
	CFsmBase	StateChart	MSM.favor_runtime_speed	MSM.favor_compile_time	QFsm.FavorExecutionSpeed	QFsm.FavorCompilationTime	QFsm.FavorDebugSize
compilation	1.13s	1.90s	10.30s	8.32s	21.79s	1.52s	2.35s
size	51K	63K	225K	253K	89K	21K	70K
strip-size	35K	31K	33K	47K	10K	8K	37K
execution	0.18s	0.12s	0.01s	0.02s	0.00s	0.01s	0.05s
valgrind	26/26 (449b)	1,000,014/1,000,014 (24,000,204b)	14/14 (646b)	122/122 (38,662b)	12/12 (102b)	12/12 (102b)	235/235 (4,718b)




CFsmBase
statechart
msm.favor_runtime_speed
msm.favor_compile_time
QFsm.FavorExecutionSpeed
QFsm.FavorCompilationTime
QFsm.FavorDebugSize
g++44 -m32 -O2 -DNDEBUG/test_complex_1000000
25
20
15
10
5
0
compilation


CFsmBase
statechart
msm.favor_runtime_speed
msm.favor_compile_time
QFsm.FavorExecutionSpeed
QFsm.FavorCompilationTime
QFsm.FavorDebugSize
g++44 -m32 -O2 -DNDEBUG/test_complex_1000000
300
250
200
150
100
50
0
size


CFsmBase
statechart
msm.favor_runtime_speed
msm.favor_compile_time
QFsm.FavorExecutionSpeed
QFsm.FavorCompilationTime
QFsm.FavorDebugSize
g++44 -m32 -O2 -DNDEBUG/test_complex_1000000
50
45
40
35
30
25
20
15
10
5
strip-size


CFsmBase
statechart
msm.favor_runtime_speed
msm.favor_compile_time
QFsm.FavorExecutionSpeed
QFsm.FavorCompilationTime
QFsm.FavorDebugSize
g++44 -m32 -O2 -DNDEBUG/test_complex_1000000
0.18
0.16
0.14
0.12
0.1
0.08
0.06
0.04
0.02
0
execution

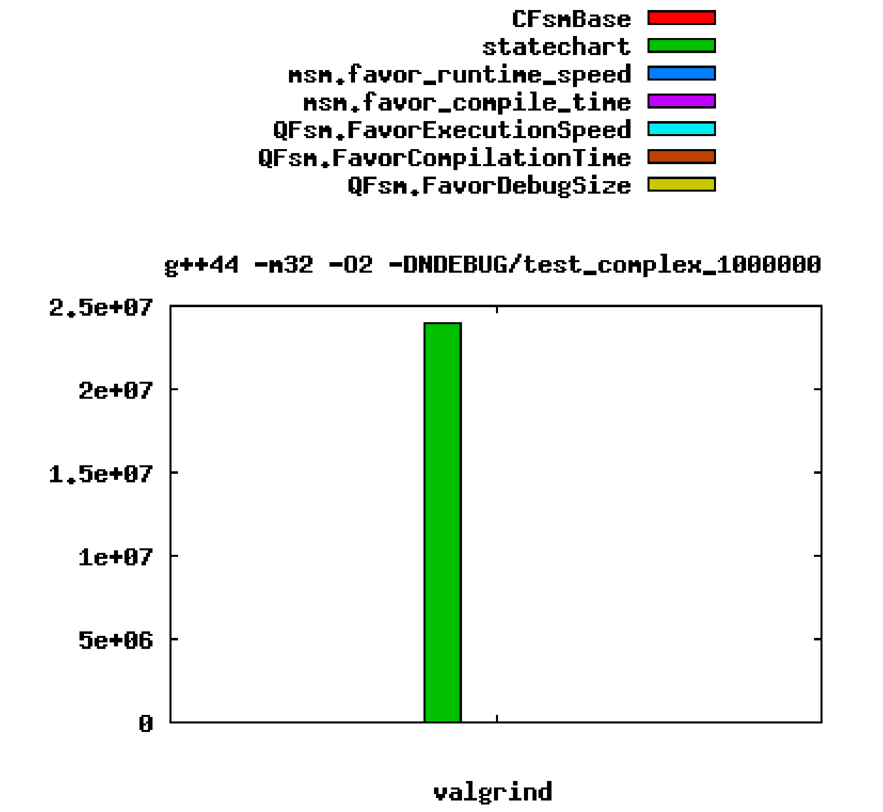
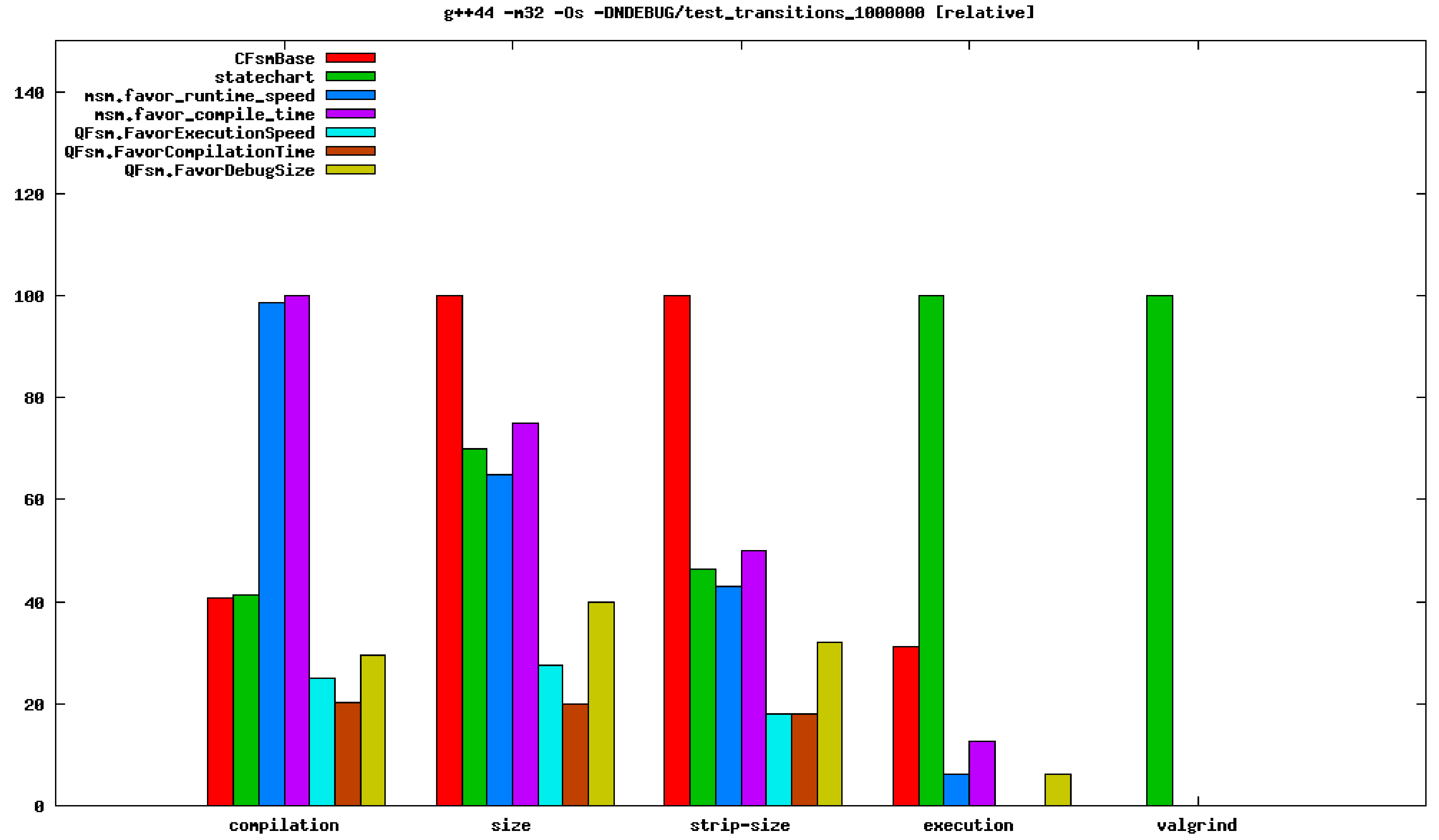
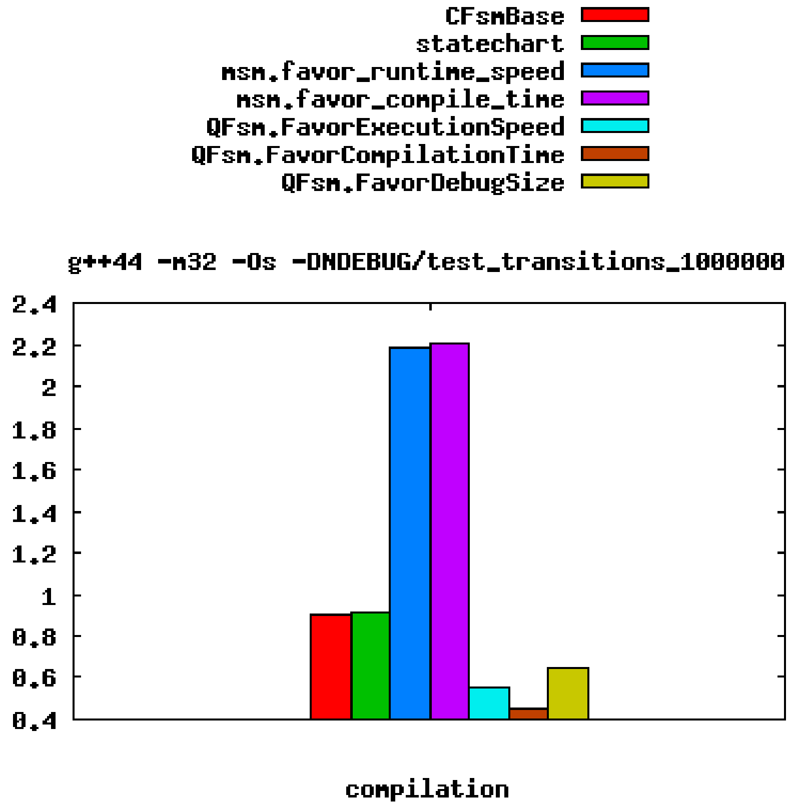
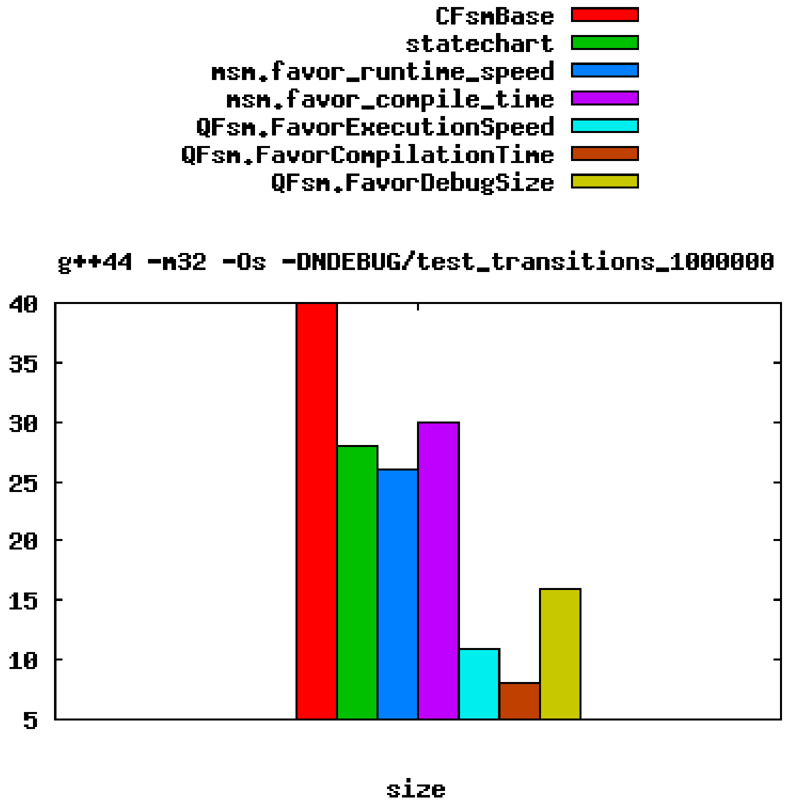

CFsmBase
statechart
msm.favor_runtime_speed
msm.favor_compile_time
QFsm.FavorExecutionSpeed
QFsm.FavorCompilationTime
QFsm.FavorDebugSize
g++44 -m32 -O2 -DNDEBUG/test_complex_1000000
2.5e+07
2e+07
1.5e+07
1e+07
5e+06
0
valgrind

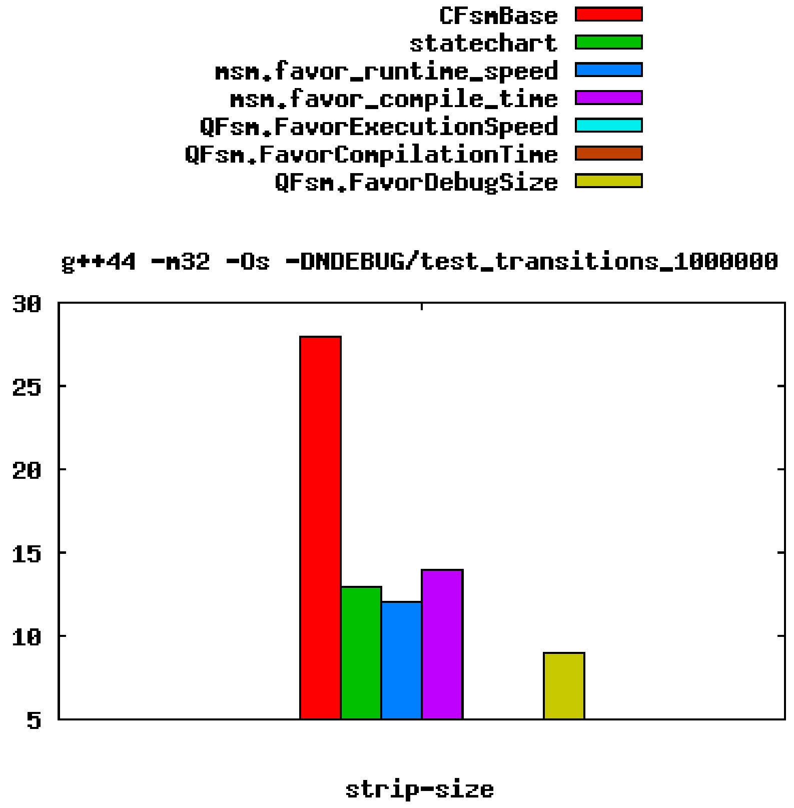
Table 2.9: ”server” [54c084f], g++44 -m32 -Os -DNDEBUG/test transitions 1000000

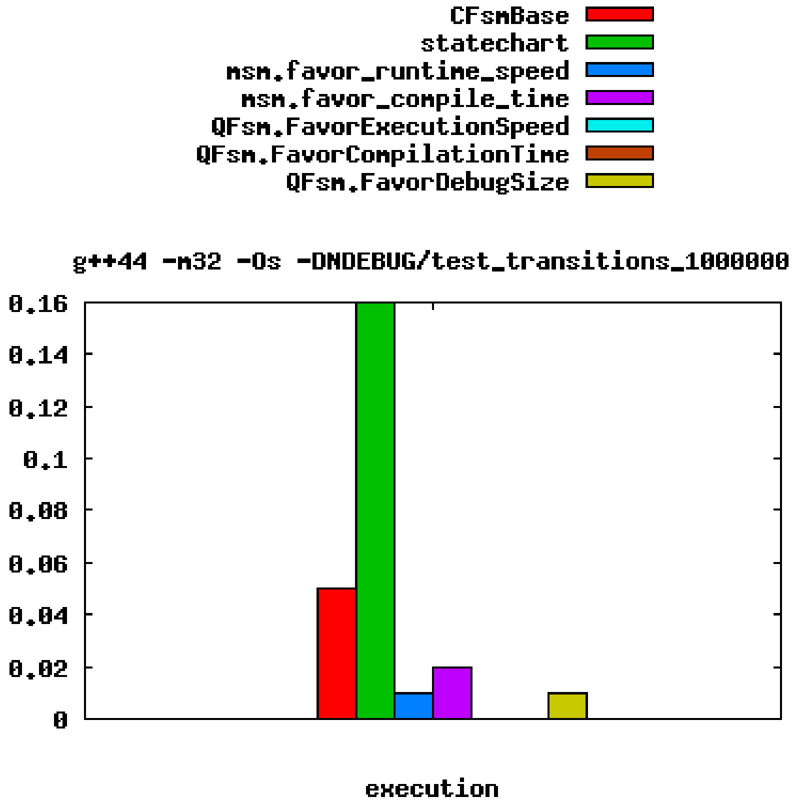
	CFsmBase	StateChart	MSM.favor_runtime_speed	MSM.favor_compile_time	QFsm.FavorExecutionSpeed	QFsm.FavorCompilationTime	QFsm.FavorDebugSize
compilation	0.90s	0.91s	2.18s	2.21s	0.55s	0.45s	0.65s
size	40K	28K	26K	30K	11K	8K	16K
strip-size	28K	13K	12K	14K	5K	5K	9K
execution	0.05s	0.16s	0.01s	0.02s	0.00s	0.00s	0.01s
valgrind	9/9 (138b)	1,000,004/1,000,004 (24,000,064b)	4/4 (561b)	10/10 (2,673b)	2/2 (17b)	2/2 (17b)	16/16 (241b)




CFsmBase
statechart
msm.favor_runtime_speed
msm.favor_compile_time
QFsm.FavorExecutionSpeed
QFsm.FavorCompilationTime
QFsm.FavorDebugSize
g++44 -m32 -Os -DNDEBUG/test_transitions_1000000
2.4
2.2
2
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
compilation


CFsmBase
statechart
msm.favor_runtime_speed
msm.favor_compile_time
QFsm.FavorExecutionSpeed
QFsm.FavorCompilationTime
QFsm.FavorDebugSize
g++44 -m32 -Os -DNDEBUG/test_transitions_1000000
40
35
30
25
20
15
10
5
size


CFsmBase
statechart
msm.favor_runtime_speed
msm.favor_compile_time
QFsm.FavorExecutionSpeed
QFsm.FavorCompilationTime
QFsm.FavorDebugSize
g++44 -m32 -Os -DNDEBUG/test_transitions_1000000
30
25
20
15
10
5
strip-size


CFsmBase
statechart
msm.favor_runtime_speed
msm.favor_compile_time
QFsm.FavorExecutionSpeed
QFsm.FavorCompilationTime
QFsm.FavorDebugSize
g++44 -m32 -Os -DNDEBUG/test_transitions_1000000
0.16
0.14
0.12
0.1
0.08
0.06
0.04
0.02
0
execution

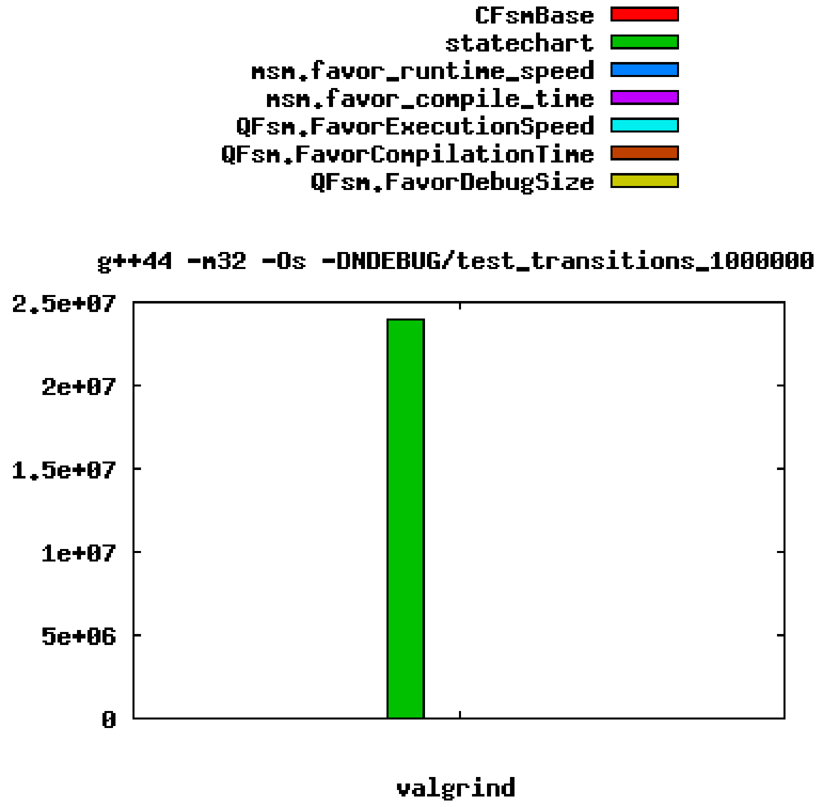
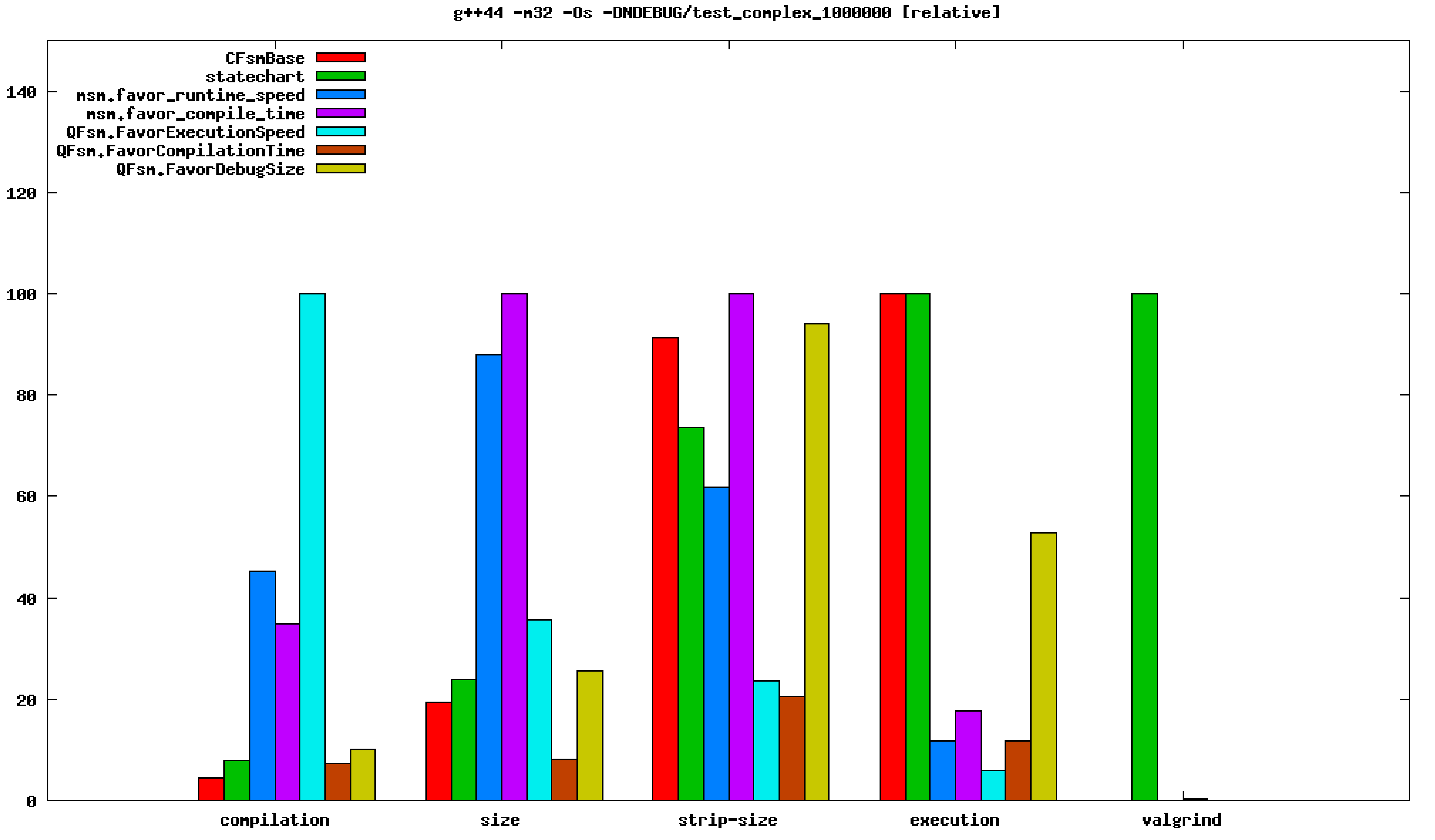
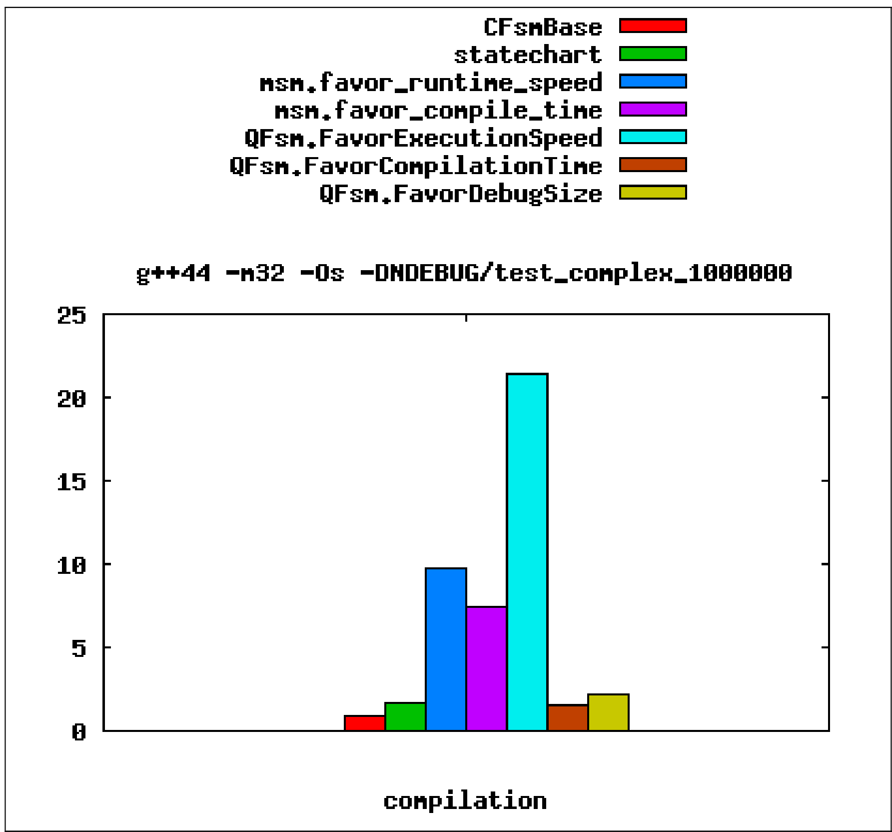
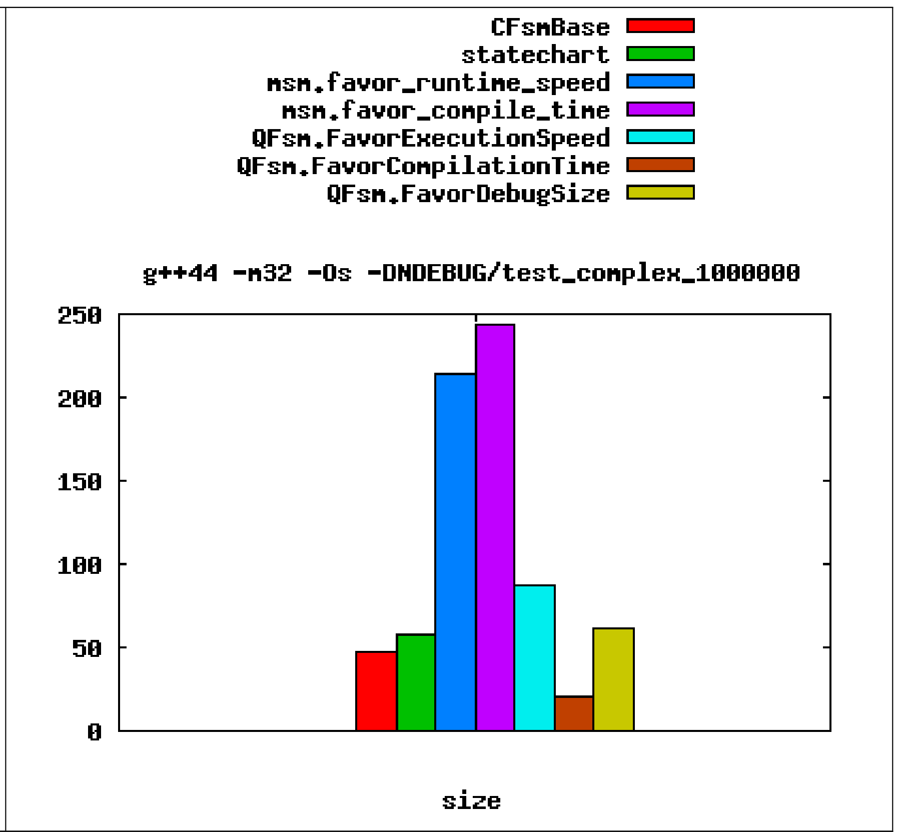

CFsmBase
statechart
msm.favor_runtime_speed
msm.favor_compile_time
QFsm.FavorExecutionSpeed
QFsm.FavorCompilationTime
QFsm.FavorDebugSize
g++44 -m32 -Os -DNDEBUG/test_transitions_1000000
2.5e+07
2e+07
1.5e+07
1e+07
5e+06
0
valgrind

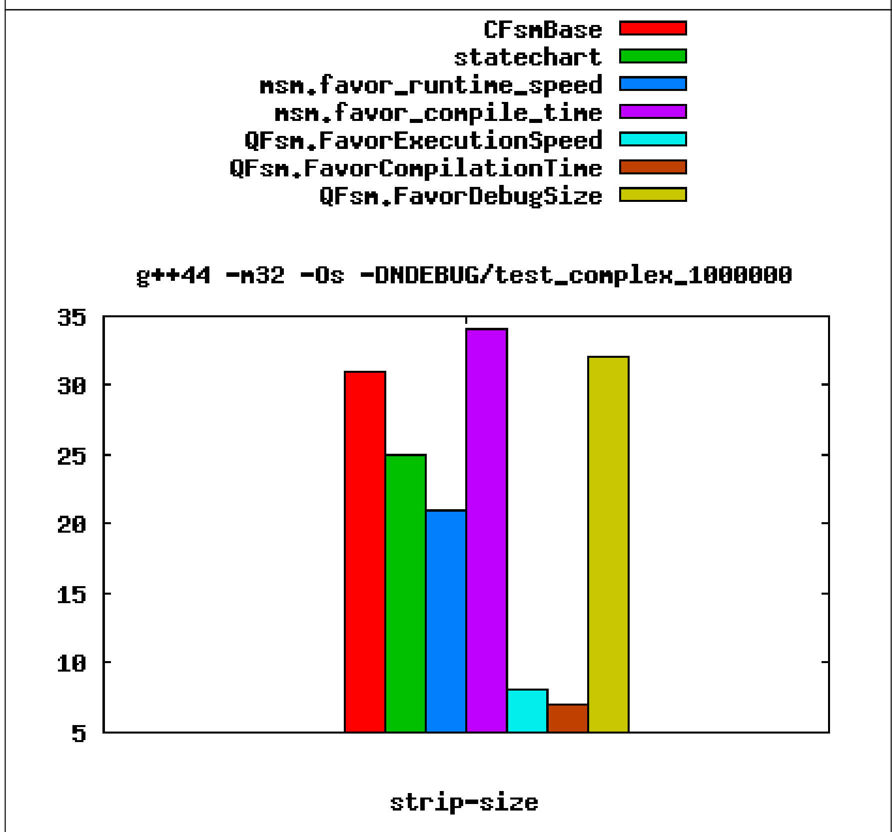
Table 2.12: "server" [54c084f], g++44 -m32 -Os -DNDEBUG/test complex 1000000

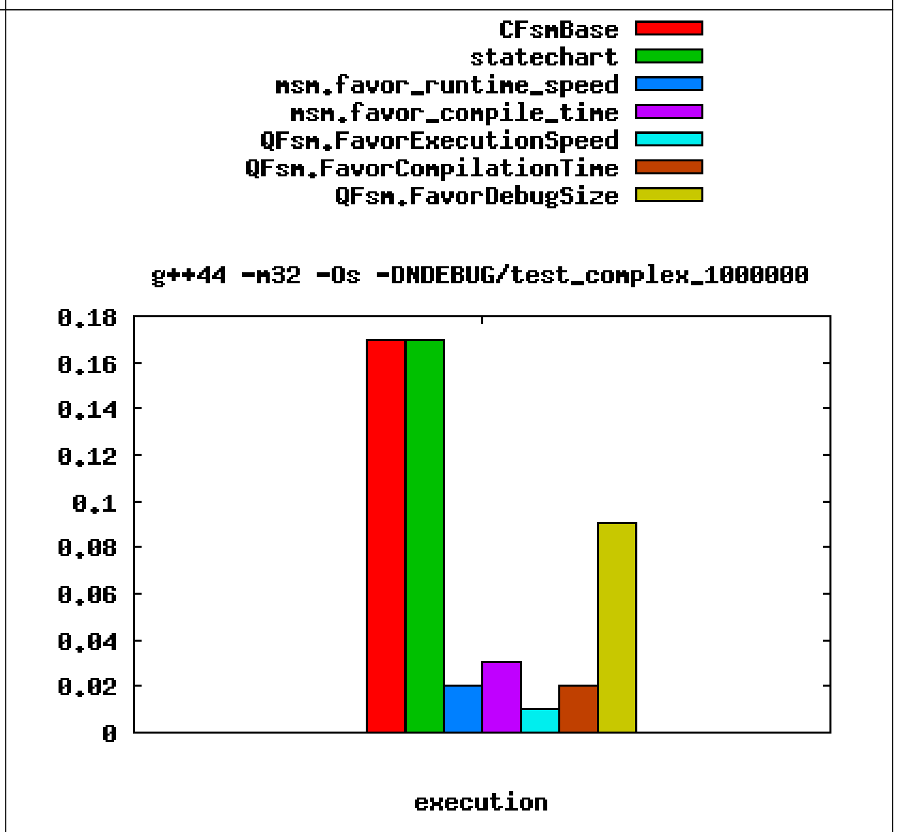
	CFsmBase	StateChart	MSM.favor_runtime_speed	MSM.favor_compile_time	QFsm.FavorExecutionSpeed	QFsm.FavorCompilationTime	QFsm.FavorDebugSize
compilation	0.96s	1.66s	9.70s	7.49s	21.42s	1.58s	2.19s
size	47K	58K	214K	243K	87K	20K	62K
strip-size	31K	25K	21K	34K	8K	7K	32K
execution	0.17s	0.17s	0.02s	0.03s	0.01s	0.02s	0.09s
valgrind	26/26 (449b)	1,000,014/1,000,014 (24,000,204b)	14/14 (646b)	122/122 (38,662b)	12/12 (102b)	12/12 (102b)	235/235 (4,718b)




CFsmBase
statechart
msm.favor_runtime_speed
msm.favor_compile_time
QFsm.FavorExecutionSpeed
QFsm.FavorCompilationTime
QFsm.FavorDebugSize
g++44 -m32 -Os -DNDEBUG/test_complex_1000000
25
20
15
10
5
0
compilation


CFsmBase
statechart
msm.favor_runtime_speed
msm.favor_compile_time
QFsm.FavorExecutionSpeed
QFsm.FavorCompilationTime
QFsm.FavorDebugSize
g++44 -m32 -Os -DNDEBUG/test_complex_1000000
250
200
150
100
50
0
size


CFsmBase
statechart
msm.favor_runtime_speed
msm.favor_compile_time
QFsm.FavorExecutionSpeed
QFsm.FavorCompilationTime
QFsm.FavorDebugSize
g++44 -m32 -Os -DNDEBUG/test_complex_1000000
35
30
25
20
15
10
5
strip-size


CFsmBase
statechart
msm.favor_runtime_speed
msm.favor_compile_time
QFsm.FavorExecutionSpeed
QFsm.FavorCompilationTime
QFsm.FavorDebugSize
g++44 -m32 -Os -DNDEBUG/test_complex_1000000
0.18
0.16
0.14
0.12
0.1
0.08
0.06
0.04
0.02
0
execution

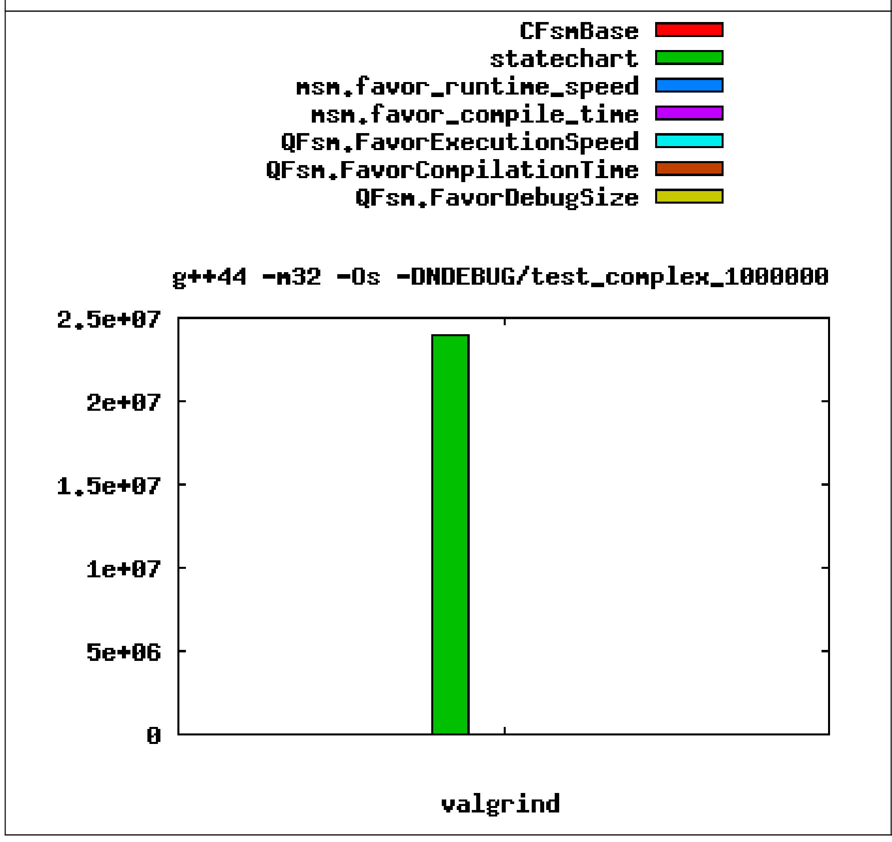
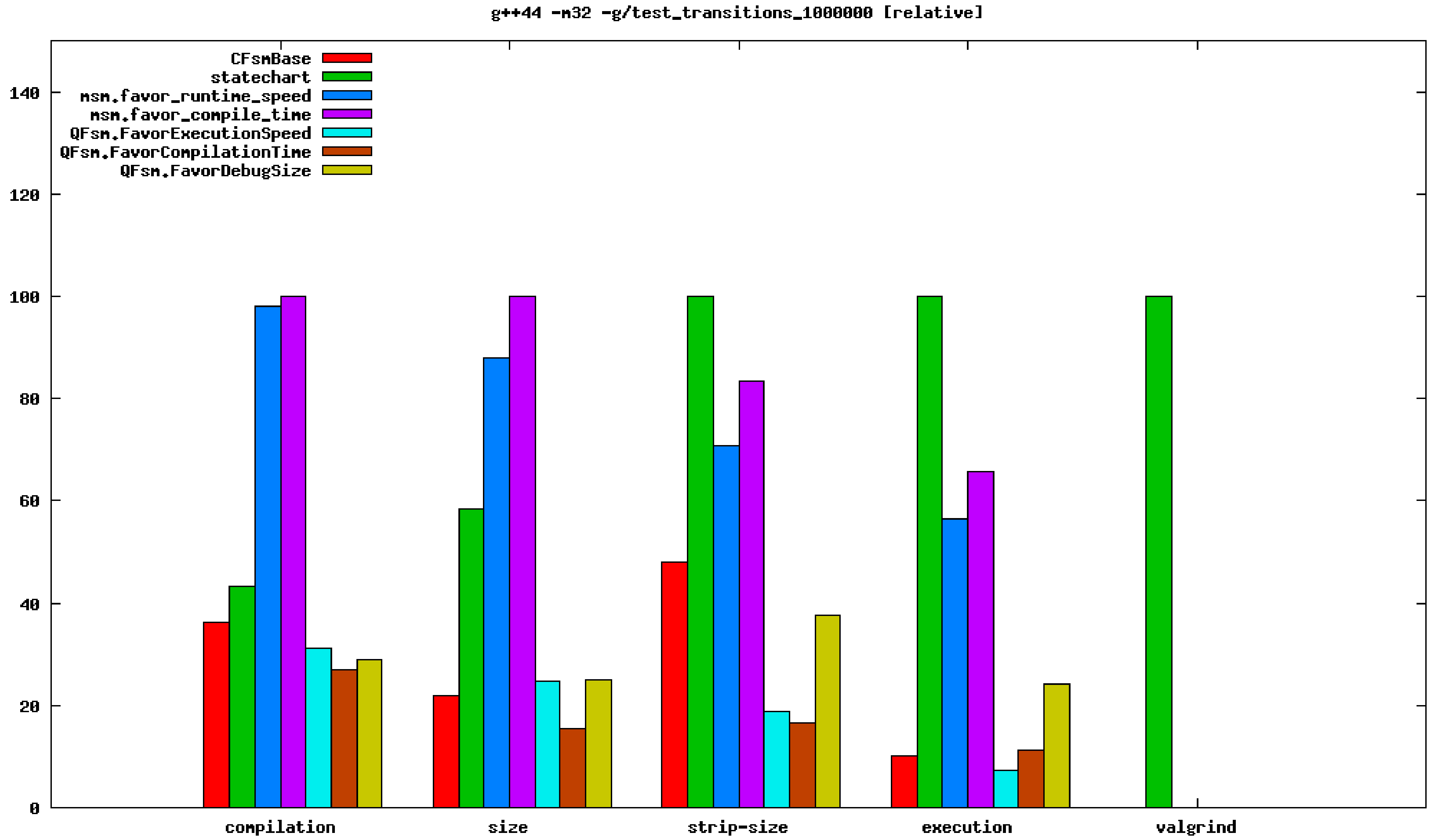
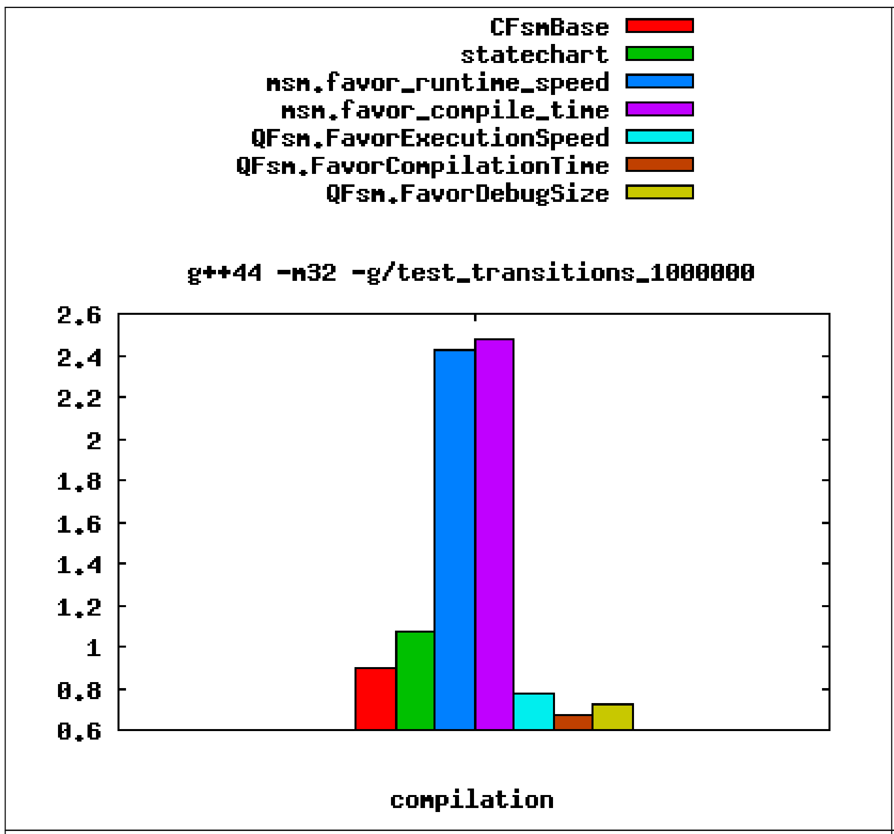
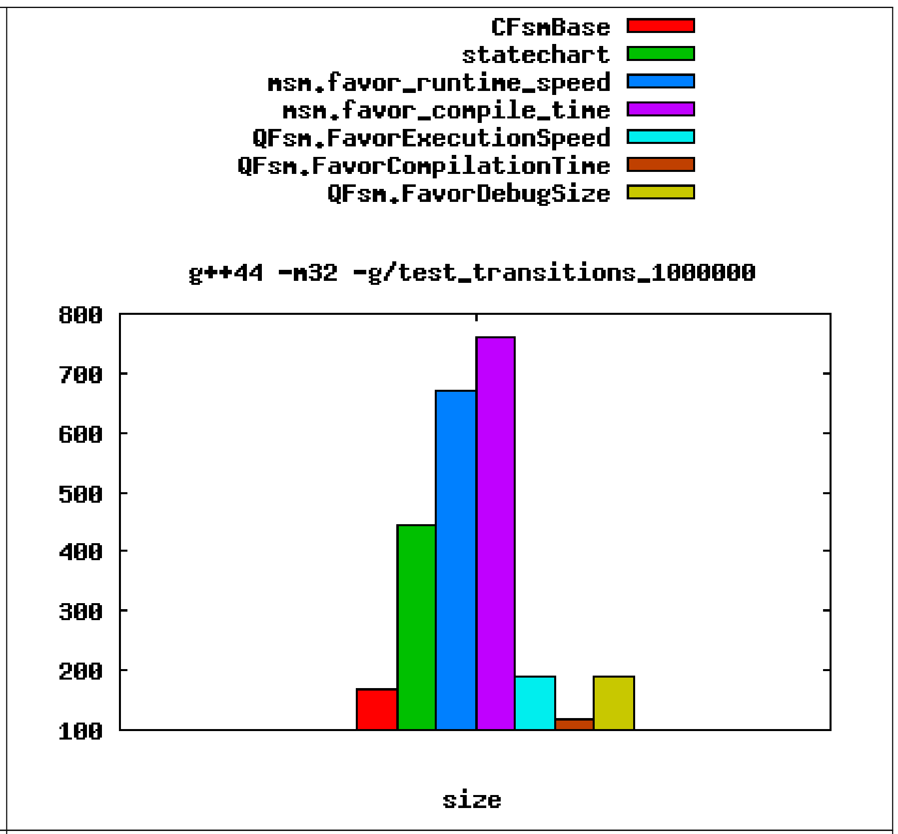

CFsmBase
statechart
msm.favor_runtime_speed
msm.favor_compile_time
QFsm.FavorExecutionSpeed
QFsm.FavorCompilationTime
QFsm.FavorDebugSize
g++44 -m32 -Os -DNDEBUG/test_complex_1000000
2.5e+07
2e+07
1.5e+07
1e+07
5e+06
0
valgrind

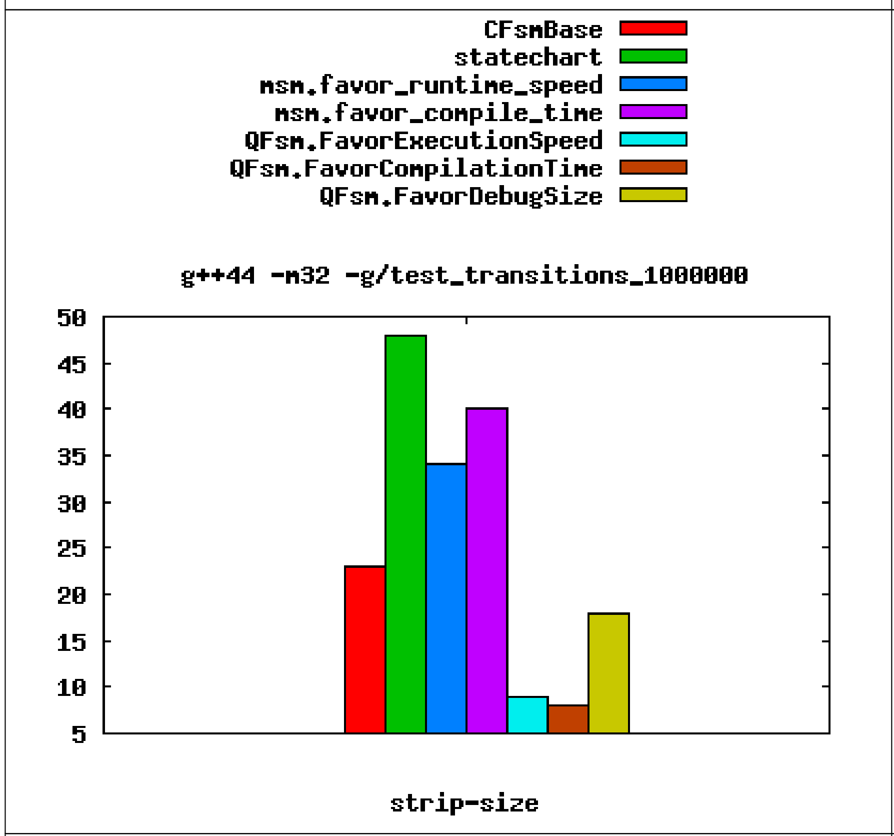
Table 2.15: "server" [54c084f], g++44 -m32 -g/test transitions 1000000

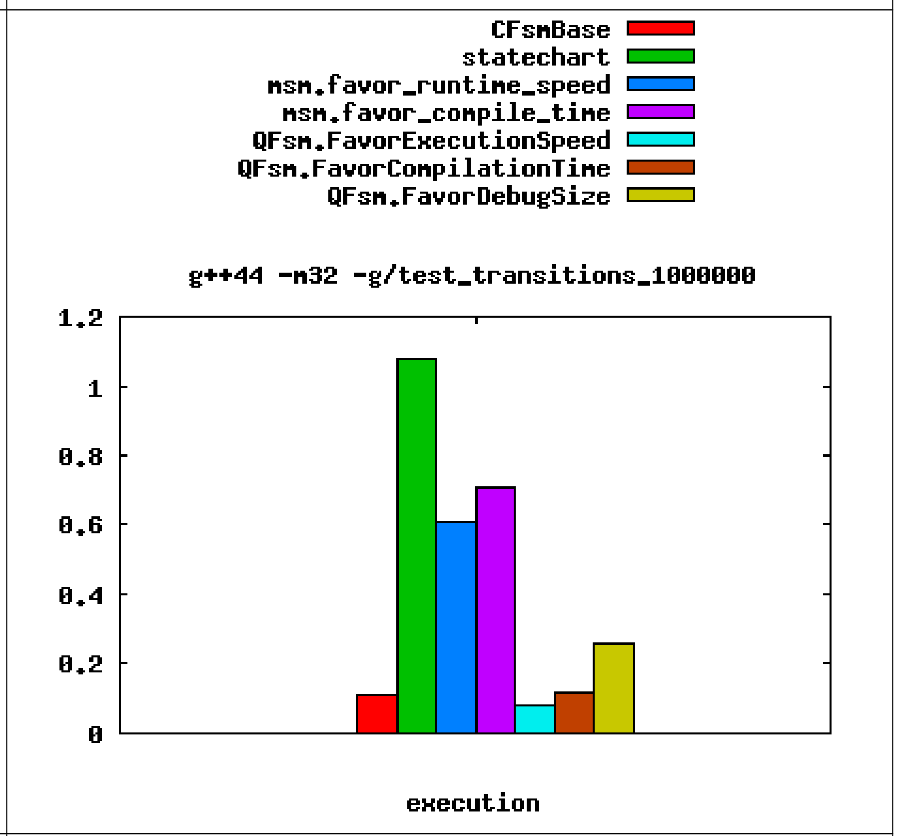
	CFsmBase	StateChart	MSM.favor_runtime_speed	MSM.favor_compile_time	QFsm.FavorExecutionSpeed	QFsm.FavorCompilationTime	QFsm.FavorDebugSize
compilation	0.90s	1.07s	2.43s	2.48s	0.77s	0.67s	0.72s
size	167K	445K	670K	762K	188K	117K	191K
strip-size	23K	48K	34K	40K	9K	8K	18K
execution	0.11s	1.08s	0.61s	0.71s	0.08s	0.12s	0.26s
valgrind	9/9 (138b)	1,000,004/1,000,004 (24,000,064b)	4/4 (561b)	10/10 (2,673b)	2/2 (17b)	2/2 (17b)	16/16 (241b)




CFsmBase
statechart
msm.favor_runtime_speed
msm.favor_compile_time
QFsm.FavorExecutionSpeed
QFsm.FavorCompilationTime
QFsm.FavorDebugSize
g++44 -m32 -g/test_transitions_1000000
2.6
2.4
2.2
2
1.8
1.6
1.4
1.2
1
0.8
0.6
compilation


CFsmBase
statechart
msm.favor_runtime_speed
msm.favor_compile_time
QFsm.FavorExecutionSpeed
QFsm.FavorCompilationTime
QFsm.FavorDebugSize
g++44 -m32 -g/test_transitions_1000000
800
700
600
500
400
300
200
100
size


CFsmBase
statechart
msm.favor_runtime_speed
msm.favor_compile_time
QFsm.FavorExecutionSpeed
QFsm.FavorCompilationTime
QFsm.FavorDebugSize
g++44 -m32 -g/test_transitions_1000000
50
45
40
35
30
25
20
15
10
5
strip-size


CFsmBase
statechart
msm.favor_runtime_speed
msm.favor_compile_time
QFsm.FavorExecutionSpeed
QFsm.FavorCompilationTime
QFsm.FavorDebugSize
g++44 -m32 -g/test_transitions_1000000
1.2
1
0.8
0.6
0.4
0.2
0
execution

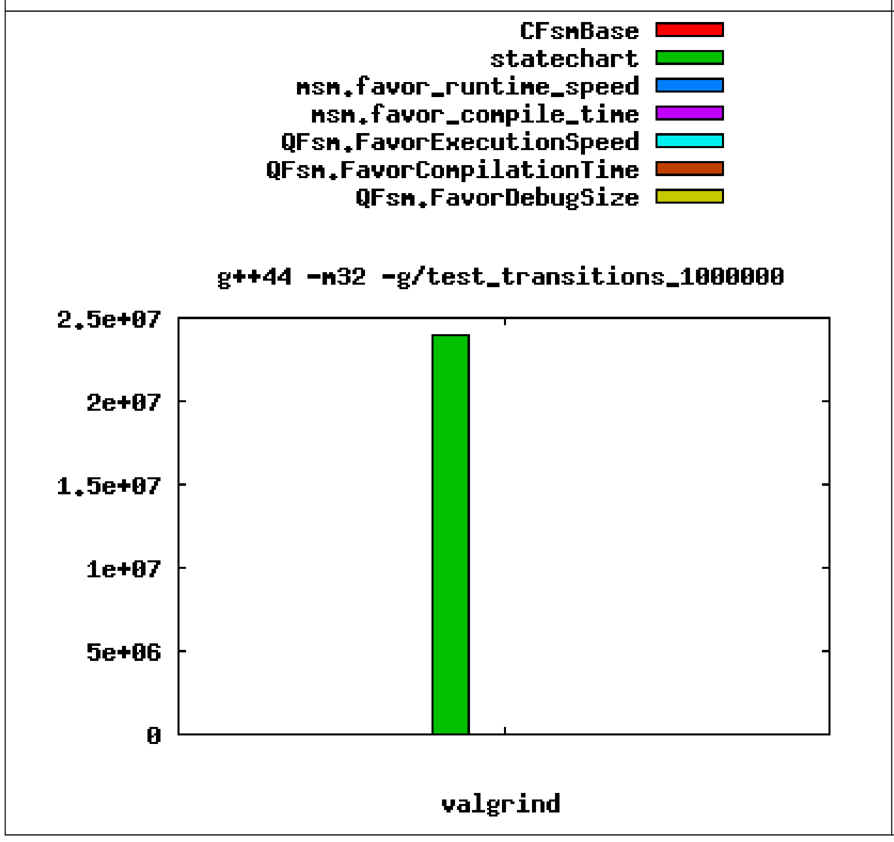
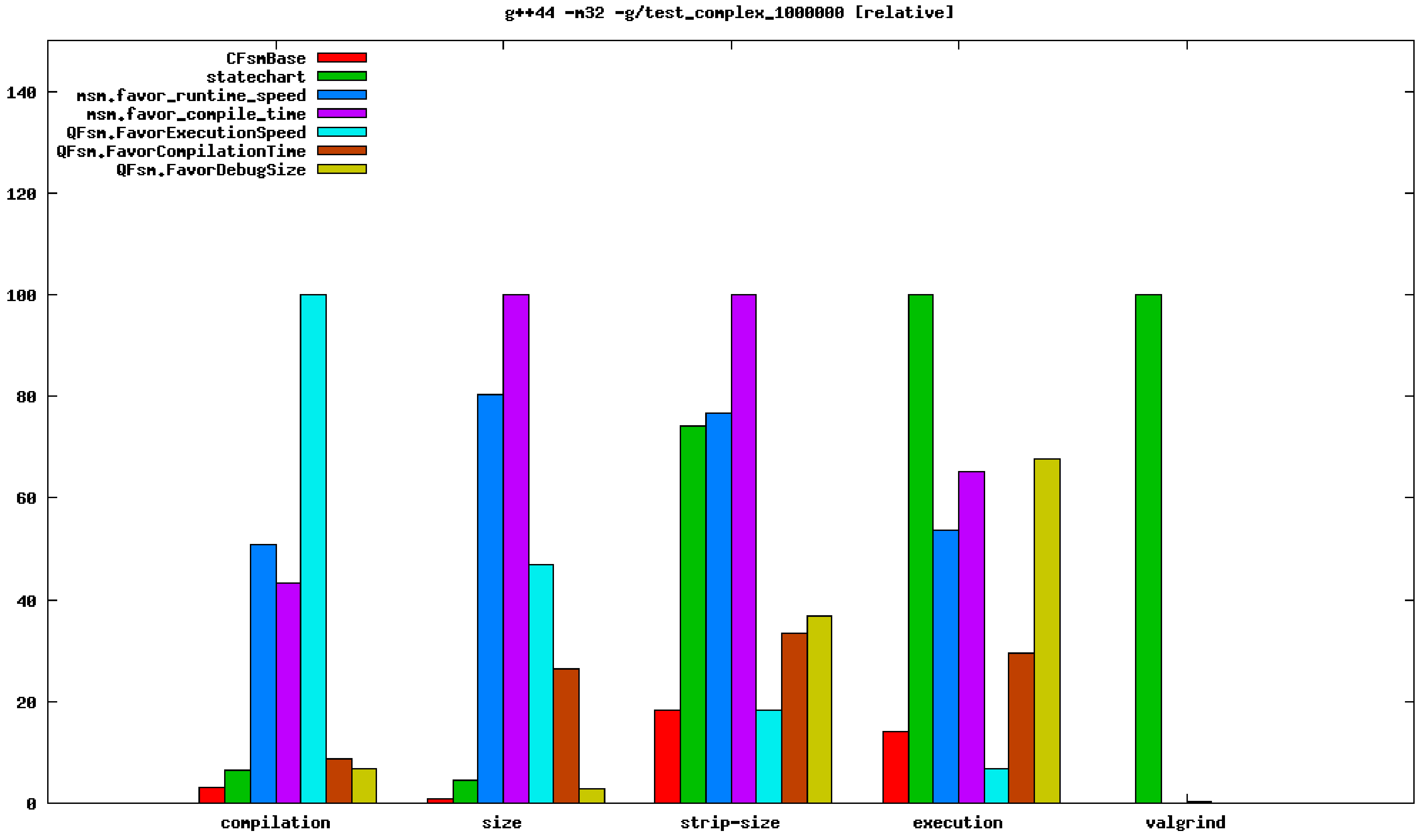
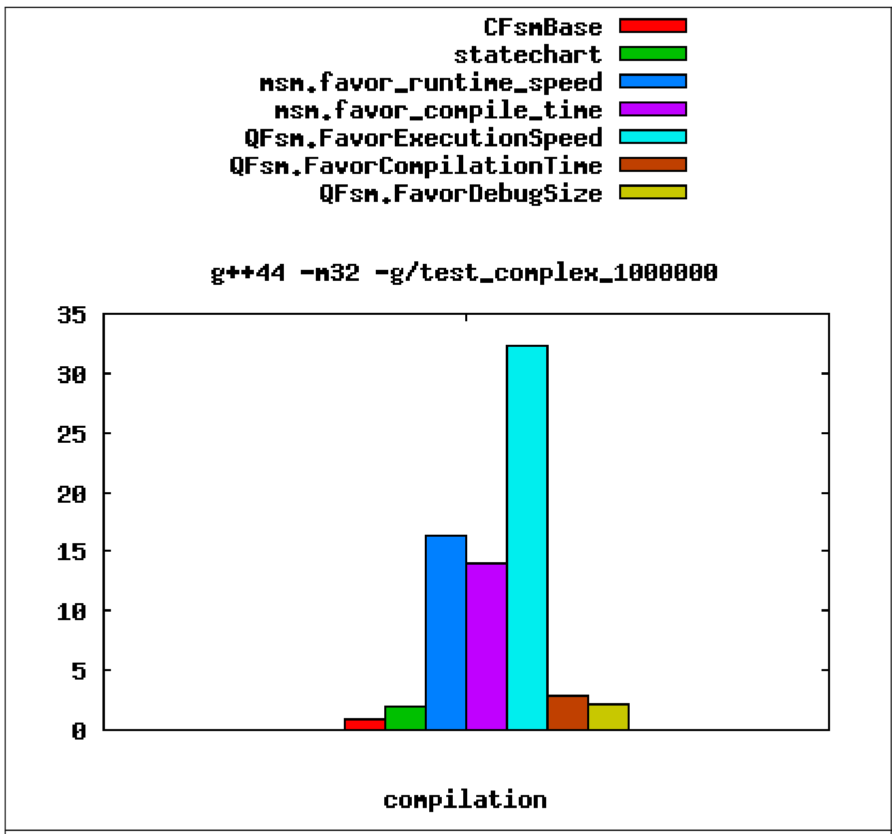
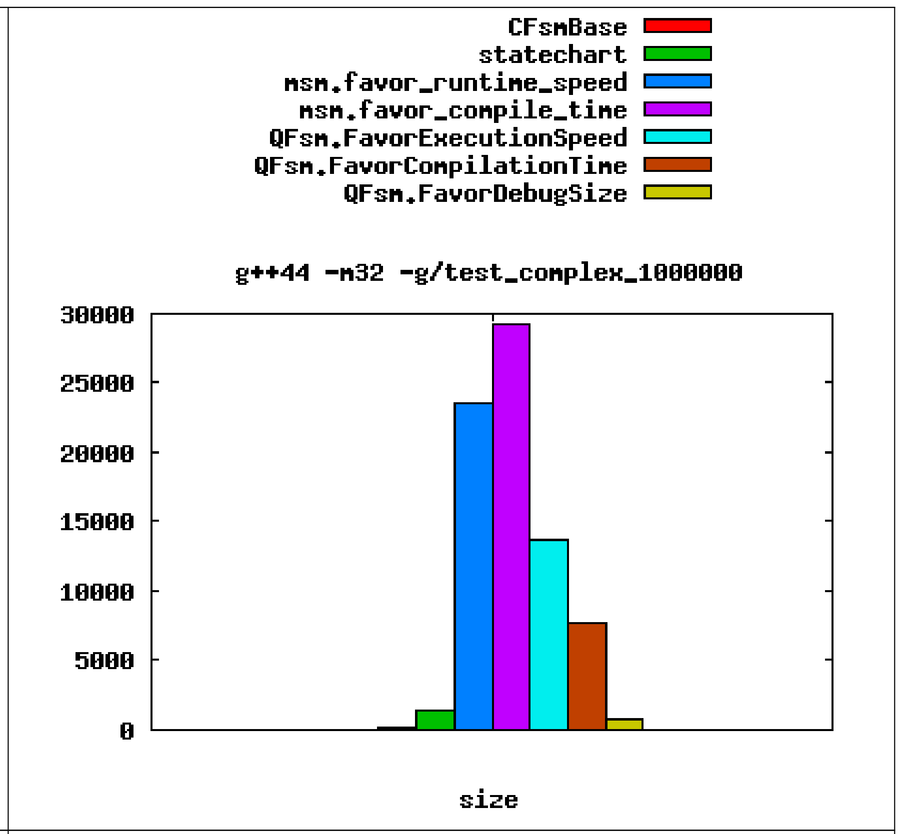

CFsmBase
statechart
msm.favor_runtime_speed
msm.favor_compile_time
QFsm.FavorExecutionSpeed
QFsm.FavorCompilationTime
QFsm.FavorDebugSize
g++44 -m32 -g/test_transitions_1000000
2.5e+07
2e+07
1.5e+07
1e+07
5e+06
0
valgrind

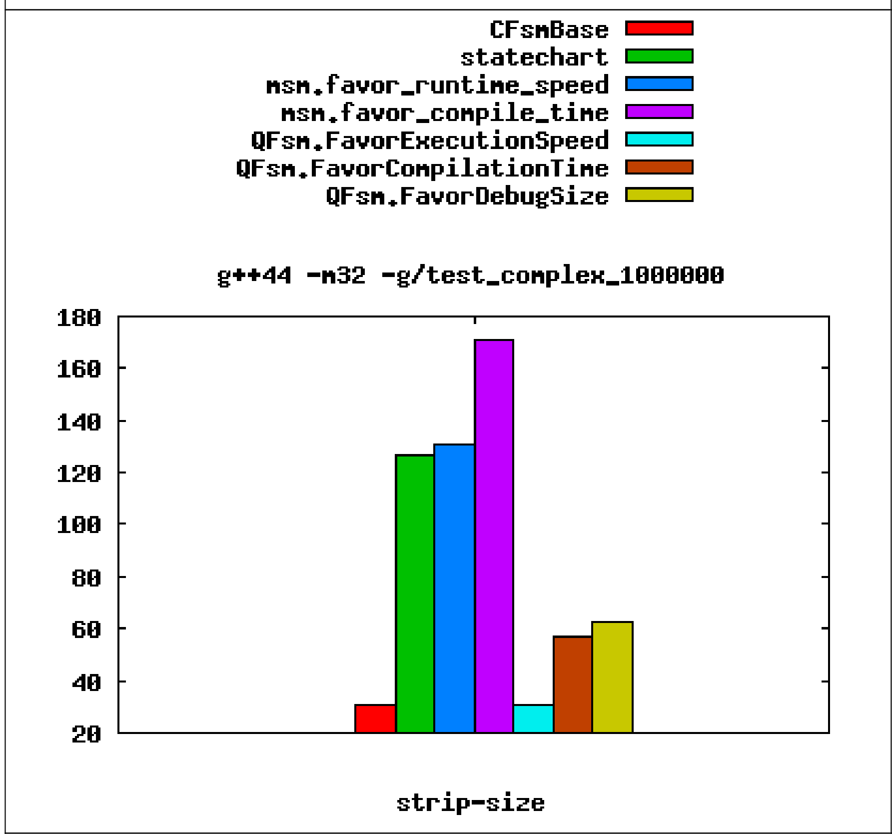
Table 2.18: "server" [54c084f], g++44 -m32 -g/test complex 1000000

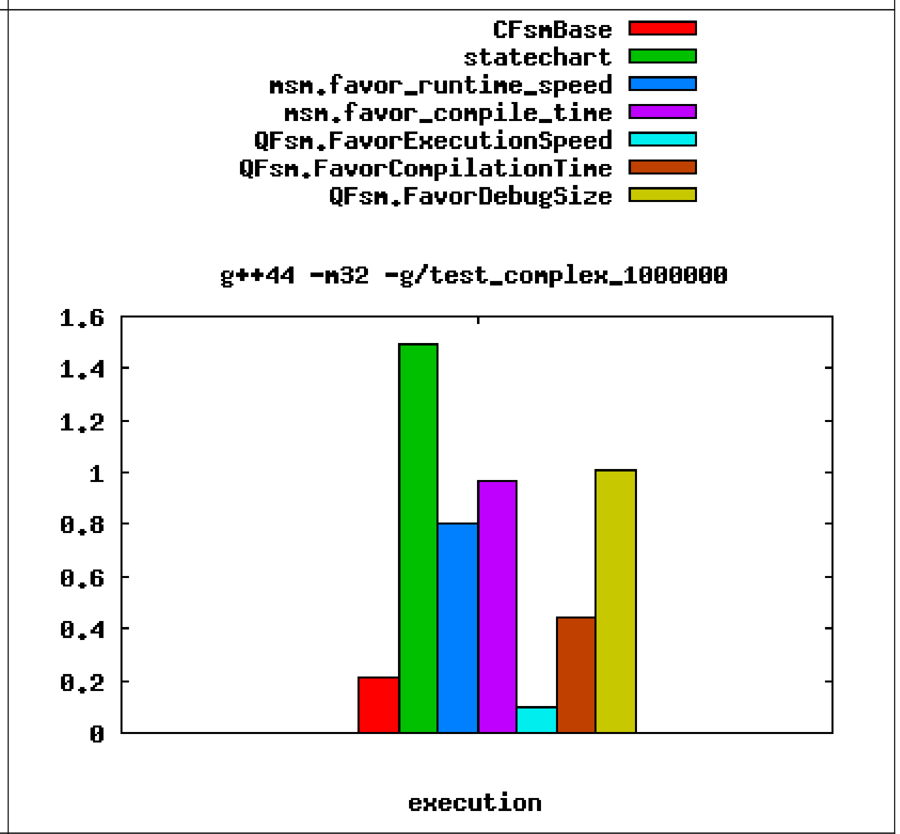
	CFsmBase	StateChart	MSM.favor_runtime_speed	MSM.favor_compile_time	QFsm.FavorExecutionSpeed	QFsm.FavorCompilationTime	QFsm.FavorDebugSize
compilation	0.98s	2.06s	16.37s	13.95s	32.26s	2.79s	2.22s
size	208K	1323K	23490K	29254K	13685K	7720K	843K
strip-size	31K	127K	131K	171K	31K	57K	63K
execution	0.21s	1.49s	0.80s	0.97s	0.10s	0.44s	1.01s
valgrind	26/26 (449b)	1,000,014/1,000,014 (24,000,204b)	14/14 (646b)	122/122 (38,662b)	12/12 (102b)	12/12 (102b)	235/235 (4,718b)

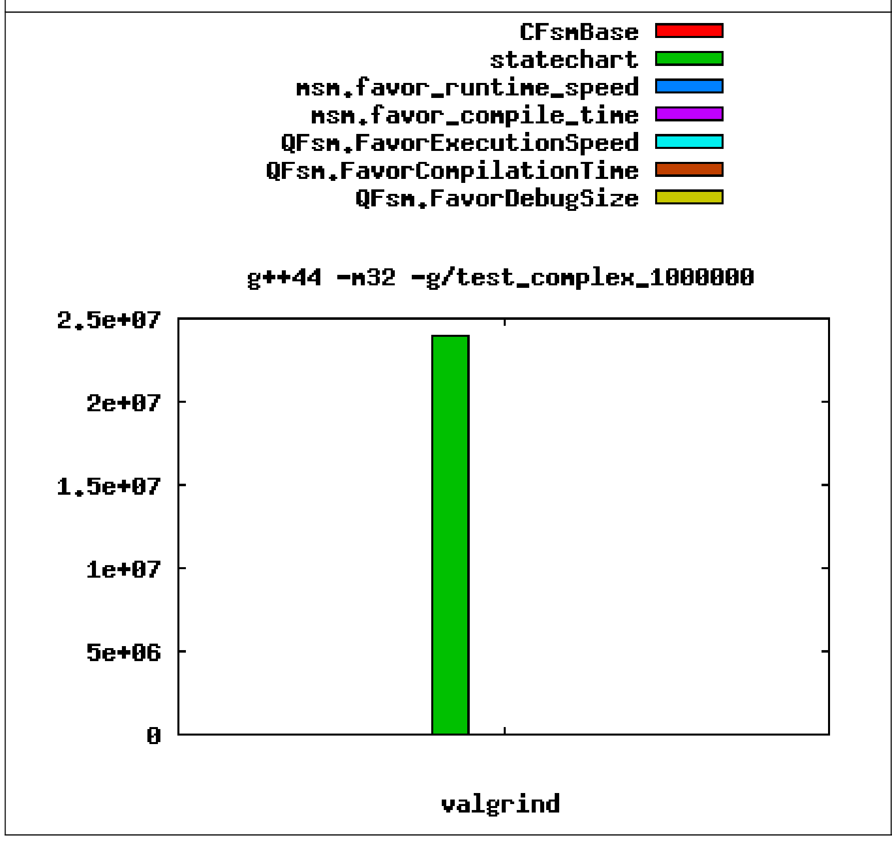



CFsmBase
statechart
msm.favor_runtime_speed
msm.favor_compile_time
QFsm.FavorExecutionSpeed
QFsm.FavorCompilationTime
QFsm.FavorDebugSize
g++44 -m32 -g/test_complex_1000000
35
30
25
20
15
10
5
0
compilation


CFsmBase
statechart
msm.favor_runtime_speed
msm.favor_compile_time
QFsm.FavorExecutionSpeed
QFsm.FavorCompilationTime
QFsm.FavorDebugSize
g++44 -m32 -g/test_complex_1000000
30000
25000
20000
15000
10000
5000
0
size


CFsmBase
statechart
msm.favor_runtime_speed
msm.favor_compile_time
QFsm.FavorExecutionSpeed
QFsm.FavorCompilationTime
QFsm.FavorDebugSize
g++44 -m32 -g/test_complex_1000000
180
160
140
120
100
80
60
40
20
strip-size


CFsmBase
statechart
msm.favor_runtime_speed
msm.favor_compile_time
QFsm.FavorExecutionSpeed
QFsm.FavorCompilationTime
QFsm.FavorDebugSize
g++44 -m32 -g/test_complex_1000000
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
execution


CFsmBase
statechart
msm.favor_runtime_speed
msm.favor_compile_time
QFsm.FavorExecutionSpeed
QFsm.FavorCompilationTime
QFsm.FavorDebugSize
g++44 -m32 -g/test_complex_1000000
2.5e+07
2e+07
1.5e+07
1e+07
5e+06
0
valgrind

Chapter 3

Summary

Document presents comparison between different C++ Finite State Machine frameworks. Tested frameworks vary in their terms of functionality, approach and concept. Frameworks were compared due to compilation speed, execution time, debug size, strip size, memory consumption, functionality and UML compatibility. Also different compilers and compilation flags were taken into account. Results shows that declarative concept might be as efficient as imperative concept while preserving at the same transparency of the state machine, as well as easy to test code. Tests also indicate that there is no one way to implement FSM framework which meet all the given criteria. Therefore frameworks like msm, QFsm have the advantage that the policy may be adjusted to better meet some of the requirements. In most cases compilation time seems to be longer in proportion to execution time. This does not apply to QFsm.FavorCompilationTime for which execution speed is kept in most tests as well as compilation time. QFsm.FavorExecutionSpeed and msm in case of compilation time seems to be far away after opponents. Execution speed shows that msm, QFsm.FavorExecutionSpeed, QFsm.FavorCompilationTime are the fastest. StateChart, QFsm.FavorDebugSize are about 10 times slower. Debug size is huge in msm and QFsm.FavorExecutionSpeed and the remaining frameworks keep the debug size within reasonable limits. Release (strip) size is the smallest in QFsm.FavorCompilationTime. About 2 times bigger in QFsm.FavorExecutionSpeed, 3 times bigger in msm, statechart. Memory consumption meets the requirements in all frameworks besides the statechart framework.

Chapter 4

References

1. David Abrahams, Aleksey Gurtovoy. "C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond" 2004.

2. Andrei Alexandrescu. "Modern C++ Design: Generic Programming and Design Patterns Applied" Boston, MA: Addison-Wesley, 2001.

3. Robert Martin. "UML Tutorial: Finite State Machines." C++ Report, June 1998. http://www.objectmentor.com/resources/articles/umlfsm.pdf.

4. David Lafreniere. "State Machine Design in C++." C++ Report, May 2000. http://www.cuj.com/documents/s=8039/cuj0005lafrenie/.

Chapter 5

Revision History

Version	Date	Modified by	Status	Accepted by	Main changes
0.1	25.09.2011	Krzysztof Jusiak	-	-	initial version
0.2	28.09.2011	Krzysztof Jusiak	-	-	corrections after review
1.0	15.11.2011	Krzysztof Jusiak	-	-	bug fix

List of Figures

List of Tables