Evaluating Computing Systems Using Fault-Injection and RAS Metrics and Models

Rean Griffith
Thesis Proposal
February 28th 2007
Outline

- Background (Goal, Motivation)
- Problem
- Requirements (Big Picture View)
- Hypotheses
- Solution Part I – Fault Injection via Kheiron
- Solution Part II – RAS-Models + 7U-evaluation
- Accomplishments
- Timeline
- Expected Contributions And Future Work
Goal

- A methodology for evaluating computing systems based on their reliability, availability and serviceability properties.
Why Bother?

- We understand speed (very well)
  - We use speed as our primary evaluation measure
- But... fast computers fail and so do slower ones
- Users demand that computing systems are also:
  - Reliable, Highly available and Serviceable (easy to manage, repair and recover)
- But...
  - Faster != More Reliable
  - Faster != More Available
  - Faster != More Serviceable
- How do we evaluate RAS-properties? We need other measures to draw conclusions on “better”. 
Wait a minute…

- Haven’t we been here before?
  - 70’s – Fault-tolerant Computing (FTC).
  - 80’s – Dependable Systems and Networks (DSN).

- What have we learned so far?
  - FTC – Fault Avoidance, Fault Masking via Redundancy, N-Versions etc.
  - DSN – Reliability & Availability via Robustness.
  - AC – Feedback architectures, 4 sub-areas of focus (self-configuration, self-healing, self-optimizing, self-protecting)
Quick Terminology

- **Reliability**
  - Number or frequency of client interruptions

- **Availability**
  - A function of the rate of failure/maintenance events and the speed of recovery

- **Serviceability**
  - A function of the number of service-visits, their duration and associated costs
More Terms...

- **Error**
  - Deviation of system state from correct service state

- **Fault**
  - Hypothesized cause of an error

- **Fault Model**
  - Set of faults the system is expected to respond to

- **Remediation**
  - Process of correcting a fault (detect, diagnose, repair)

- **Failure**
  - Delivered service violates an environmental constraint e.g. SLA or policy
Requirements

- How do we study a system’s RAS-properties?
  - Construct a representative fault-model
  - Build fault-injection tools to induce the faults in the fault-model
  - Study the impact of faults on the target system with any remediation mechanisms turned off then on
  - Evaluate the efficacy of any existing remediation mechanisms via their impact on SLAs, policies, etc.
  - Evaluate the expected impact of yet-to-be added remediation mechanisms (if possible)
Hypotheses

- Runtime adaptation is a reasonable technology for implementing efficient and flexible fault-injection tools.
- RAS-models, represented as Continuous Time Markov Chains (CTMCs), are a reasonable framework for analyzing system failures, remediation mechanisms and their impact on system operation.
- RAS-models and fault-injection experiments can be used together to model and measure the RAS-characteristics of computing systems. This combination links the details of the mechanisms to the high-level goals governing the system’s operation, supporting comparisons of individual or combined mechanisms.
Spoiler...

- **Part I**
  - Kheiron: a new framework for runtime-adaptation in a variety of applications in multiple execution environments.
  - Fault-injection tools built on top of Kheiron

- **Part II**
  - System analysis using RAS-models.
  - The 7-steps (our proposed 7U-evaluation) methodology linking the analysis of individual and combined mechanisms to the high-level goals governing the system’s operation.
One “What” & Three “Why’s”

- What is runtime-adaptation?
- Why runtime-adaptation?
- Why build fault-tools using this technology?
- Why build our own fault tools?
Four answers...

- **What is runtime-adaptation?**
  - Ability to make changes to applications while they execute.

- **Why runtime-adaptation?**
  - Flexible, preserves availability, manages performance

- **Why build fault-tools using this technology?**
  - Fine-grained interaction with application internals.

- **Why build our own fault tools?**
  - Different fault-model/focus from robustness oriented tools like FAUMachine, Ferrari, Ftape, Doctor, Xception, FIST, MARS, Holodeck and Jaca.
Kheiron Features

- Able to make changes in running .NET, Java and Compiled C-applications.
- Low overhead.
- Transparent to both the application and the execution environments.
- No need for source-code access.
- No need for specialized versions of the execution environments.
3 implementations of Kheiron
- Kheiron/CLR, Kheiron/JVM and Kheiron/C

Key observation
- All software runs in an execution environment (EE), so use it to facilitate adapting the applications it hosts.

Two kinds of EEs
- Unmanaged (Processor + OS e.g. x86 + Linux)
- Managed (CLR, JVM)

For this to work the EE needs to provide 4 facilities…
## EE-Support

<table>
<thead>
<tr>
<th>EE Facilities</th>
<th>Unmanaged Execution Environment</th>
<th>Managed Execution Environment</th>
</tr>
</thead>
</table>
| **Program tracing**           | ptrace, /proc                   | JVM 5.x                      | CLR 1.1
| **Program control**           | Trampolines + Dyninst           | Bytecode rewriting           | MSIL rewriting
| **Execution unit metadata**   | .symtab, .debug sections        | Classfile constant pool + bytecode | Assembly, type & method metadata + MSIL
| **Metadata augmentation**     | N/A for compiled C-programs     | Custom classfile parsing & editing APIs + JVMTI RedefineClasses | IMetaDataImport, IMetaDataEmit APIs

15
SampleMethod( args ) [throws NullPointerException]
<room for prolog>
push args
call _SampleMethod( args ) [throws NullPointerException]
{ try{…} catch (IOException ioe){…} } // Source view of _SampleMethod’s body
<room for epilog>
return value/void
Kheiron/CLR & Kheiron/JVM Fault-Rewrite

```java
public void someMethod()
{
    call StatsCop.methodEnter( "someMethod" ) // profile method enter
    call FaultManager.injectFault( "someMethod" ) // lookup fault to inject
    call _someMethod(); // call original implementation of someMethod
    call StatsCop.methodExit( "someMethod" ) // profile method exit
}
```
# Kheiron/C Operation

<table>
<thead>
<tr>
<th>Mutator</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kheiron/C</td>
<td>void foo( int x, int y)</td>
</tr>
<tr>
<td>Dyninst API</td>
<td>{</td>
</tr>
<tr>
<td>Dyninst Code</td>
<td>int z = 0;</td>
</tr>
<tr>
<td>ptrace/procfs</td>
<td>}</td>
</tr>
</tbody>
</table>

- Snippets
- C/C++ Runtime Library

```c
void foo( int x, int y)
{
    int z = 0;
}
```
Kheiron/C – Prologue Example

```c
static int i = 0;
void SomeFunc() {
    i = i + 10;
}
```

```c
08049100 <_Z8SomeFuncv>:
  08049100: 55    push %ebp
  08049101: 89 e5  mov %esp,%ebp
  08049103: 83 05 c4 6c 05 08 0a  addl $0xa,0x08056cc4
  0804910a: 5d    pop %ebp
  0804910b: c3    ret

runtime transformation

08049100 <_Z8SomeFuncv>:
  08049100: e9 a0 99 68 40  jmp 0x406899a0 (jump to trampoline)
  08049103: c4 6c 05 08 0a  instruction mangled by trampoline insertion
  0804910a: 5d    pop %ebp (next valid instruction)
  0804910b: c3    ret (return to calling function)

406899a0 <trampoline>:
  save CPU registers
  // inserted assembly from snippet e.g. a function call
  restore CPU registers
  jump to saved/relocated instructions

<saved/relocated instructions>:
  55    push %ebp
  89 e5  mov %esp,%ebp
  83 05 c4 6c 05 08 0a  addl $0xa,0x08056cc4
  e9 0a 91 04 08  jmp 0x804910a (jump to next valid instruction)
Kheiron/CLR & Kheiron/JVM Feasibility

Kheiron/CLR Overheads when no adaptations active

Kheiron/JVM Overheads when no adaptations active
Kheiron/C Feasibility

Performance comparison SciMark - normalized to w/o Dyninst - simple jump into adaptation library

Kheiron/C Overheads when no adaptations active
Sophisticated Runtime Adaptations

- Transparent hot-swap of the job scheduler component in the Alchemi Enterprise Grid Computing System using Kheiron/CLR
  - Kheiron/CLR performs a component hot-swap without disrupting work in the grid or crashing the CLR.

- Supporting the selective emulation of compiled C-functions using Kheiron/C
  - Kheiron/C loads the STEM x86 emulator into the address space of a target program and causes selected functions to run under emulation rather than on the real processor.
Part I Summary

- Kheiron supports contemporary managed and unmanaged execution environments.
- Low-overhead (<5% performance hit).
- Transparent to both the application and the execution environment.
- Access to application internals
  - Class instances (objects) & Data structures
  - Components, Sub-systems & Methods
- Capable of sophisticated adaptations.
- Fault-injection tools built with Kheiron leverage all its capabilities.
Outline

- Background (Goal, Motivation)
- Problem
- Requirements (Big Picture View)
- Hypotheses
- Solution Part I – Fault Injection via Kheiron
- Solution Part II – RAS-Models + 7U-evaluation
- Accomplishments
- Timeline
- Expected Contributions And Future Work
Target System for RAS-study

- N-Tier web application
  - TPC-W web-application & Remote Browser Emulators
  - Resin 3.0.22 application server & web server (running Sun Hotspot JVM 1.5)
  - MySQL 5.0.27
  - Linux 2.4.18 kernel

- Fault model
  - Device driver faults injected using SWIFI device driver fault-injection tools
  - Memory-leaks injected using Kheiron/JVM-based tool
## Expected Fault-Model Coverage

<table>
<thead>
<tr>
<th>Fault Category</th>
<th>Target</th>
<th>Remediation</th>
</tr>
</thead>
</table>
| Memory Leak                          | Web-application server/Web-application classes                | System reboot (reactive)  
Application-server restart (reactive)  
Application-server restart (preventative) – To Be Added |
| 28 possible device driver faults      | Operating system kernel                                        | System reboot (reactive)  
Nooks driver recovery (reactive)        |
Analytical Tools

- **RAS-models** (Continuous Time Markov Chains)
  - Based on Reliability Theory.
  - Capable of analyzing individual or combined RAS-enhancing mechanisms.
  - Able to reason about perfect and imperfect mechanisms.
  - Able to reason about yet-to-be-added mechanisms.

- **7U-Evaluation methodology**
  - Combines fault-injection experiments and RAS-models and metrics to evaluate systems.
  - Establish a link between the mechanisms and their impact on system goals/constraints.
Reliability Theory Techniques Used

- **Continuous Time Markov Chains (CTMCs)**
  - Collection of states \((S_0, \ldots, S_n)\) connected by arcs.
  - Arcs between states represent transition rates.
  - State transitions can occur at any instant.

- **Markov assumptions**
  - \(P(X_n = i_n | X_0 = i_0, \ldots, X_{n-1} = i_{n-1}) = P(X_n = i_n | X_{n-1} = i_{n-1})\)

- **Birth-Death Processes**
  - Nearest-neighbor state-transitions only.

- **Non-Birth-Death Processes**
  - Nearest-neighbor state-transition restriction relaxed.
A: Fault-Free Operation

- TPC-W run takes ~24 minutes

<table>
<thead>
<tr>
<th>client-side</th>
<th>server-side</th>
<th>success rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>interactions: 3973</td>
<td>memory requests: 1848</td>
<td>memory: 100%</td>
</tr>
<tr>
<td></td>
<td>memory requests granted: 1848</td>
<td>execution: n/a</td>
</tr>
<tr>
<td></td>
<td>fork requests: 0</td>
<td>reads: 99.5563%</td>
</tr>
<tr>
<td></td>
<td>forks performed: 0</td>
<td>writes: 100%</td>
</tr>
<tr>
<td></td>
<td>read requests: 3,498,678</td>
<td>opens: 100%</td>
</tr>
<tr>
<td></td>
<td>reads performed: 3,483,154</td>
<td>closes: 100%</td>
</tr>
<tr>
<td></td>
<td>write requests: 22,369</td>
<td></td>
</tr>
<tr>
<td></td>
<td>writes performed: 22,369</td>
<td></td>
</tr>
<tr>
<td></td>
<td>open requests: 18,476</td>
<td></td>
</tr>
<tr>
<td></td>
<td>opens performed: 18,476</td>
<td></td>
</tr>
<tr>
<td></td>
<td>close requests: 18,560</td>
<td></td>
</tr>
<tr>
<td></td>
<td>closes performed: 18,560</td>
<td></td>
</tr>
</tbody>
</table>

Table 3: Metrics for Configuration A, Fault-Free Run
B: Memory Leak Scenario

- 1 Failure every 8 hours (40 runs = 16 hours of activity)
- Resin restarts under low memory condition. Restart takes ~47 seconds and resolves the issue each time.
B: Memory Leak Analysis

- Birth-Death process with 2 states, 2 parameters:
  - $S_0$ – UP state, system working
  - $S_1$ – DOWN state, system restarting
  - $\lambda_{\text{failure}} = 1/8$ hrs
  - $\mu_{\text{repair}} = 47$ seconds

- Assumptions
  - Perfect repair

- Results
  - Limiting/steady-state availability = 99.838%
  - Downtime per year = 866 minutes

- Is this good or bad?
  - Two 9’s availability

<table>
<thead>
<tr>
<th>Availability Guarantee</th>
<th>Max Downtime Per Year</th>
<th>Expected Penalties</th>
</tr>
</thead>
<tbody>
<tr>
<td>99.99</td>
<td>~5 mins</td>
<td>(866 - 5)*$p$</td>
</tr>
<tr>
<td>99.9</td>
<td>~53 mins</td>
<td>(866 - 53)*$p$</td>
</tr>
<tr>
<td>99.0</td>
<td>~526 mins</td>
<td>(866 - 526)*$p$</td>
</tr>
<tr>
<td>99</td>
<td>~5256 mins</td>
<td>$0$</td>
</tr>
</tbody>
</table>

Table 3. Expected SLA Penalties for Configuration B
C: Driver Faults w/o Nooks – Analysis

- Birth-Death process with 2 states, 2 parameters:
  - $S_0$ – UP state, system working
  - $S_1$ – DOWN state, system restarting
  - $\lambda_{\text{failure}} = 4/8 \text{ hrs}$
  - $\mu_{\text{repair}} = 82 \text{ seconds}$

- Assumptions
  - Perfect repair

- Results
  - Limiting/steady-state availability = 98.87%
  - Downtime per year = 5924 minutes

- Is this good or bad?
  - Less than Two 9’s availability

<table>
<thead>
<tr>
<th>Availability Guarantee</th>
<th>Max Downtime Per Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>99.999</td>
<td>~5 mins</td>
</tr>
<tr>
<td>99.9</td>
<td>~53 mins</td>
</tr>
<tr>
<td>99.9</td>
<td>~526 mins</td>
</tr>
<tr>
<td>99.0</td>
<td>~5256 mins</td>
</tr>
</tbody>
</table>
D: Driver Faults w/Nooks – Analysis

- Birth-Death process with 3 states, 4 parameters:
  - $S_0$ – UP state, system working
  - $S_1$ – UP state, recovering failed driver
  - $S_2$ – DOWN state, system reboot
  - $\lambda_{\text{driver\_failure}} = 4/8$
  - $\mu_{\text{nooks\_recovery}} = 4,093$ microseconds
  - $\mu_{\text{reboot}} = 82$ seconds
  - $c$ – coverage factor

- Assumptions
  - Imperfect Repair

- Results
  - Modest Nooks success rates needed to improve system availability.
E: Complete Fault Model – Analysis

- Birth-Death process with 4 states, 5 parameters:
  - $S_0$ – UP state, system working
  - $S_1$ – UP state, recovering failed driver
  - $S_2$ – DOWN state, system reboot
  - $S_3$ – DOWN state, Resin reboot
  - $\lambda_{\text{driver\_failure}} = 4/8 \text{ hrs}$
  - $\mu_{\text{nooks\_recovery}} = 4,093 \text{ microseconds}$
  - $\mu_{\text{reboot}} = 82 \text{ seconds}$
  - $c$ – coverage factor
  - $\lambda_{\text{memory\_leak}} = 1/8 \text{ hours}$
  - $\mu_{\text{restart\_resin}} = 47 \text{ seconds}$

- Assumptions
  - Imperfect Repair

- Results
  - Minimum downtime = 866 minutes
  - Availability limited by memory leak handling
Preventative Maintenance – Analysis

- Non-Birth-Death process with 6 states, 6 parameters:
  - $S_0$ – UP state, first stage of lifetime
  - $S_1$ – UP state, second stage of lifetime
  - $S_2$ – DOWN state, Resin reboot
  - $S_3$ – UP state, inspecting memory use
  - $S_4$ – UP state, inspecting memory use
  - $S_5$ – DOWN state, preventative restart
  - $\lambda_{2nd stage} = 1/6$ hrs
  - $\lambda_{failure} = 1/2$ hrs
  - $\mu_{restart_resin_worst} = 47$ seconds
  - $\lambda_{inspect} = \text{Rate of memory use inspection}$
  - $\mu_{inspect} = 21,627$ microseconds
  - $\mu_{restart_resin_pm} = 3$ seconds

- Results
  - Infrequent checks could have an impact.
  - Only by implementing such a scheme and running experiments would we know for sure.
Towards a RAS-Benchmark

Thought experiment

- Type 1 – No detection capabilities.
- Type 2 – Perfect detection, no diagnosis or repair.
- Type 3 – Perfect detection and diagnosis, no repair.
- Type 4 – Perfect detection, diagnosis and repair.
- Type 5 – Perfect detection, but detectors turned off.

Expected ranking

- Type 1 < Type 5 < Type 2 < Type 3 < Type 4

<table>
<thead>
<tr>
<th>macro-view</th>
<th>goodput</th>
<th>reliability, availability and serviceability</th>
<th>fault-model coverage (expected vs actual)</th>
</tr>
</thead>
<tbody>
<tr>
<td>micro-view</td>
<td>accuracy of detection, diagnosis and repair</td>
<td>speed of detection, diagnosis and repair</td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Example Metrics
7-Step Evaluation “Recipe”

- 7U-Evaluation methodology
  - Combines fault-injection experiments and RAS-models and metrics to evaluate systems.
  - Establish a link between the mechanisms and their impact on system goals/constraints.
  - Highlights the role of the environment in scoring and comparing system.
Part II Summary

- RAS-models are powerful yet flexible tools
  - Able to analyze individual and combined mechanisms.
  - Able to analyze reactive and preventative mechanisms.
  - Capable of linking the details of the mechanisms to their impact on system goals (SLAs, policies etc.)
  - Useful as design-time and post-deployment analysis-tools.

- Limitations
  - Assumption of independence makes it difficult to use them to study cascading/dependent faults.
Accomplishments To Date

- 3 papers on runtime adaptations
  - DEAS 2005 (Kheiron/CLR).
  - ICAC 2006 (Kheiron/JVM, Kheiron/C).
  - Chapter in Handbook on Autonomic Computing.

- Submission to ICAC 2007
  - Using RAS-models and Metrics to evaluate Self-Healing Systems.
<table>
<thead>
<tr>
<th>Timeline</th>
<th>Work</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jan. 2006</td>
<td>Submitted Kheiron Paper to ICAC</td>
<td>Accepted</td>
</tr>
<tr>
<td>Sep. 2006</td>
<td>Build GUI front-end for Kheiron/JVM</td>
<td>Ongoing</td>
</tr>
<tr>
<td>Oct. 2006</td>
<td>Build self-healing benchmark simulator</td>
<td>Completed</td>
</tr>
<tr>
<td>Nov. 2006</td>
<td>Build Linux-based test-bed for RAS-experiments</td>
<td>Completed</td>
</tr>
<tr>
<td>Dec. 2006</td>
<td>Run preliminary RAS-benchmarking experiments</td>
<td>Completed</td>
</tr>
<tr>
<td>Jan. 2007</td>
<td>Submit paper on initial results to ICAC 2007</td>
<td>Completed</td>
</tr>
<tr>
<td>Feb. 2007</td>
<td>Write Thesis Proposal</td>
<td>Completed</td>
</tr>
<tr>
<td>Mar. 2007</td>
<td>Port Linux 2.4 device driver fault tools to Linux 2.6</td>
<td>Ongoing</td>
</tr>
<tr>
<td>Mar. 2007</td>
<td>Write device driver fault tool for Windows XP</td>
<td>Ongoing</td>
</tr>
<tr>
<td>May. 2007</td>
<td>Write proof of concept database fault injection tool</td>
<td>Ongoing</td>
</tr>
<tr>
<td>Jun. 2007</td>
<td>Write or acquire under NDA Solaris 10 fault-injection tools</td>
<td>Ongoing</td>
</tr>
<tr>
<td>Jul. 2007</td>
<td>Build test machine for hardware &amp; software fault injection</td>
<td>Ongoing</td>
</tr>
<tr>
<td>Aug. 2007</td>
<td>Start next round of RAS-experiments (Solaris, Linux, Win32)</td>
<td>Ongoing</td>
</tr>
<tr>
<td>Jan. 2008</td>
<td>Thesis writing</td>
<td></td>
</tr>
<tr>
<td>Aug. 2008</td>
<td>Thesis defense</td>
<td></td>
</tr>
</tbody>
</table>
Expected Contributions

- Contributions towards a representative fault-model for computing systems that can be reproduced using fault-injection tools.
- A suite of runtime fault-injection tools to complement existing software-based and hardware-based fault-injection tools.
- A survey of the RAS-enhancing mechanisms (or lack thereof) in contemporary operating systems and application servers.
- Analytical techniques that can be used at design-time or post-deployment time.
- A RAS-benchmarking methodology based on practical fault-injection tools and rigorous analytical techniques.
Thank You...

- Questions?
- Comments?
- Queries?
Backup Slides
Kheiron Architecture from 10,000ft
How Kheiron Works

- Attaches to programs while they run or when they load.
- Interacts with programs while they run at various points of their execution.
  - Augments type definitions and/or executable code
  - Needs metadata – rich metadata is better
- Interposes at method granularity, inserting new functionality via method prologues and epilogues.
- Control can be transferred into/out of adaptation library logic
- Control-flow changes can be done/un-done dynamically
# System Operation

<table>
<thead>
<tr>
<th>Time period/execution event</th>
<th>Unmanaged/Native Applications (C-Programs)</th>
<th>Managed Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application start</td>
<td>Attach Kheiron, augment methods</td>
<td>Load Kheiron/JVM</td>
</tr>
<tr>
<td>Module load</td>
<td>No real metadata to manipulate</td>
<td>Augment type definition, augment module metadata, bytecode rewrite</td>
</tr>
<tr>
<td>Method invoke/entry</td>
<td>Transfer control to adaptation logic</td>
<td>Transfer control to adaptation logic</td>
</tr>
<tr>
<td>Method JIT</td>
<td>n/a</td>
<td>No explicit notifications</td>
</tr>
<tr>
<td>Method exit</td>
<td>Transfer control to adaptation logic</td>
<td>Transfer control to adaptation logic</td>
</tr>
</tbody>
</table>

- **CLM 1.1**
- **CLR 1.1**
- **JVM 5.x**
- **managed Applications**
- **Unmanaged/Native Applications**
- **Module load**
- **Method entry**
- **Module JIT**
- **Method exit**
- **Augment module**
- **metadata**
- **MSIL rewrite**
- **force re-jit**
- **n/a**
- **Time period/execution event**
- **application start**
- **Module load**
- **Method invoke/entry**
- **Method JIT**
- **Method exit**
Experiments

- **Goal:** Measure the feasibility of our approach.

- Look at the impact on execution when no repairs/adaptations are active.

- Selected compute-intensive applications as test subjects (SciMark and Linpack).

- **Unmanaged experiments**
  - P4 2.4 GHz processor, 1GB RAM, SUSE 9.2, 2.6.8x kernel, Dyninst 4.2.1.

- **Managed experiments**
  - P3 Mobile 1.2 GHz processor, 1GB RAM, Windows XP SP2, Java HotspotVM v1.5 update 04.
Unmanaged Execution Environment

Metadata

- Not enough information to support type discovery and/or type relationships.
- No APIs for metadata manipulation.
- In the managed world, units of execution are self-describing.

Symbol Table Entry

typedef struct {
  Elf32_Word   st_name;
  Elf32.Addr   st_value;
  Elf32.Word   st_size;
  unsigned char st_info;
  unsigned char st_other;
  Elf32.Half   st_shndx;
} Elf32_Sym;

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>STT_NOTYPE</td>
<td>0</td>
</tr>
<tr>
<td>STT_OBJECT</td>
<td>1</td>
</tr>
<tr>
<td>STT_FUNC</td>
<td>2</td>
</tr>
<tr>
<td>STT_SECTION</td>
<td>3</td>
</tr>
</tbody>
</table>