Maggie AI Assistant Resource Management Package

Manual & Implementation Guide

This document provides a comprehensive guide to the Maggie AI Assistant resource management package, which is specifically optimized for systems utilizing AMD Ryzen 9 5900X CPUs and NVIDIA RTX 3080 GPUs.

Table of Contents

General Overview

The Maggie AI Assistant resource management package (`maggie.utils.resource`) provides a comprehensive framework for hardware detection, performance optimization, resource monitoring, and dynamic resource management. It is designed to maximize performance on systems utilizing AMD Ryzen 9 5900X CPUs and NVIDIA RTX 3080 GPUs while maintaining system stability.

The package intelligently adapts to the application's current state (IDLE, READY, ACTIVE, BUSY, etc.) to allocate appropriate resources, optimize hardware utilization, and prevent resource contention. It includes specialized optimizations for tensor operations, memory management, thread affinity, and GPU acceleration.

Key features include:

- **Hardware Detection**: Automatic identification of system components with specialized detection for AMD Ryzen 9 5900X, NVIDIA RTX 3080, and XPG D10 memory
- **Hardware-Specific Optimization**: Tailored optimizations including core affinity, tensor core utilization, and memory management
- **Resource Monitoring**: Real-time tracking of CPU, memory, and GPU utilization with configurable thresholds
- **State-Aware Resource Management**: Dynamic resource allocation based on application state
- **Proactive Warning System**: Early warnings for potential resource issues
- **Performance Profiling**: Creation of optimization profiles based on hardware configuration
- **Graceful Degradation**: Managed reduction of resource usage when thresholds are approached

Package Architecture

The resource management package consists of four primary components:

1. **HardwareDetector** (`detector.py`): Responsible for identifying system hardware and capabilities
2. **HardwareOptimizer** (`optimizer.py`): Creates and applies hardware-specific optimizations
3. **ResourceMonitor** (`monitor.py`): Tracks resource utilization and triggers warnings when thresholds are exceeded
4. **ResourceManager** (`manager.py`): Main interface that integrates the other components and provides state-aware resource management

These components work together through the following workflow:

1. The **HardwareDetector** identifies the system hardware upon initialization
2. The **HardwareOptimizer** uses this information to create hardware-specific optimization profiles
3. The **ResourceManager** applies these optimizations based on the current state
4. The **ResourceMonitor** continuously tracks resource utilization and triggers events when thresholds are exceeded
5. The **ResourceManager** responds to these events by adjusting resource allocation

Implementation Guide

Basic Integration

To integrate the resource management package into your Maggie AI application, follow these steps:

1. **Initialize the Resource Manager**

```
from maggie.utils.resource.manager import ResourceManager
from maggie.service.locator import ServiceLocator

# Initialize the resource manager with the system configuration
resource_manager = ResourceManager(config)

# Register with ServiceLocator for access by other components
ServiceLocator.register('resource_manager', resource_manager)
```

2. **Connect to State Management System**

```
# Connect to state manager if not already registered during initialization
if resource_manager.state_manager is None:
    resource_manager.setup_state_management()

# Connect to event system if not already registered
if resource_manager.event_bus is None:
    resource_manager.setup_event_system()
```

3. **Set up GPU Environment**

```
# Initialize GPU environment with optimizations
resource_manager.setup_gpu()

# Apply hardware-specific optimizations
```

```
optimizations = resource_manager.apply_hardware_specific_optimizations()
```

4. **Start Resource Monitoring**

```
# Start monitoring with specified polling interval (in seconds)
resource_manager.start_monitoring(interval=5.0)
```

Advanced Integration

For more advanced integration, you can:

1. **Register Event Listeners**

```
def resource_event_handler(event_type, event_data):
    if event_type == 'low_memory_warning':
        # Handle low memory condition
        pass
    elif event_type == 'gpu_memory_warning':
        # Handle GPU memory warning
        pass

# Register the event handler
resource_manager.register_resource_event_listener(resource_event_handler)
```

2. **Optimize for Specific States**

```
from maggie.core.state import State

# Optimize resources for a specific state
profile = resource_manager.get_state_specific_profile(State.ACTIVE)
```

3. **Request Resource Reports**

```
# Get the current resource status
status = resource_manager.get_resource_status()

# Get hardware recommendations
recommendations = resource_manager.recommend_configuration()

# Get detailed hardware report
report = resource_manager.get_hardware_report()
```

State Transition Handling

The ResourceManager is designed to automatically respond to state transitions. When the application transitions between states (e.g., from IDLE to READY), the ResourceManager will:

1. Apply state-specific resource policies
2. Adjust CPU priority

3. Manage memory allocation
4. Configure GPU memory allocation
5. Clear caches if necessary

For custom handling of specific state transitions:

```
def custom_transition_handler(transition):
    from_state = transition.from_state
    to_state = transition.to_state

    if from_state == State.ACTIVE and to_state == State.BUSY:
        # Custom handling for ACTIVE -> BUSY transition
        resource_manager.clear_gpu_memory()
```

Usage Examples

Example 1: Basic Integration with Automatic Optimization

```
from maggie.utils.resource import get_resource_manager
from maggie.utils.config.manager import ConfigManager

# Load configuration
config_manager = ConfigManager('config.yaml')
config = config_manager.load()

# Initialize resource manager
ResourceManager = get_resource_manager()
resource_manager = ResourceManager(config)

# Setup and start resource management
resource_manager.setup_gpu()
resource_manager.apply_hardware_specific_optimizations()
resource_manager.start_monitoring()

# Application code continues...
# Resource manager will automatically handle state transitions
```

Example 2: Responding to Resource Warnings

```
def handle_resource_warning(event_type, event_data):
    if event_type == 'low_memory_warning':
        percent = event_data.get('percent', 0)
        available_gb = event_data.get('available_gb', 0)
        print(f"Low memory warning: {percent:.1f}% used, {available_gb:.1f} GB available")

        # Reduce memory usage
        clear_caches()
        unload_non_essential_components()

    elif event_type == 'gpu_memory_warning':
        percent = event_data.get('percent', 0)
        print(f"GPU memory warning: {percent:.1f}% used")
```

```
        # Reduce GPU memory usage
        release_gpu_tensors()
        reduce_batch_size()

# Register the event handler
resource_manager.register_resource_event_listener(handle_resource_warning)
```

Example 3: Manually Optimizing for LLM Processing

```
def prepare_for_llm_operation():
    # Clear GPU memory before loading LLM
    resource_manager.clear_gpu_memory()

    # Get the current state and apply optimizations
    current_state = state_manager.get_current_state()
    profile = resource_manager.get_state_specific_profile(current_state)

    # Explicitly optimize for RTX 3080 if available
    gpu_info = resource_manager.hardware_info.get('gpu', {})
    if gpu_info.get('is_rtx_3080', False):
        gpu_opts = resource_manager.optimizer.optimize_for_rtx_3080()

    # Apply CPU optimizations if using Ryzen 9 5900X
    cpu_info = resource_manager.hardware_info.get('cpu', {})
    if cpu_info.get('is_ryzen_9_5900x', False):
        cpu_opts = resource_manager.optimizer.optimize_for_ryzen_9_5900x()

    # Load the LLM with optimized settings
    llm_processor.load_model()
```

Example 4: Preallocating Resources for State Transitions

```
def prepare_for_active_state():
    from maggie.core.state import State

    # Preallocate resources for ACTIVE state
    success = resource_manager.preallocate_for_state(State.ACTIVE)

    if success:
        # Resources successfully preallocated
        print("Resources preallocated for ACTIVE state")
        return True
    else:
        # Insufficient resources available
        print("Insufficient resources for ACTIVE state")
        return False

# Check if we can transition to ACTIVE state
if prepare_for_active_state():
    state_manager.transition_to(State.ACTIVE, "user_interaction")
else:
    # Handle insufficient resources
```

```
    display_resource_warning()
```

Configuration Options

The resource management package is highly configurable through the `config.yaml` file. Key configuration sections include:

CPU Configuration

```yaml
cpu:
  # Number of worker threads for parallel processing
  max_threads: 8
  # Timeout for thread operations in seconds
  thread_timeout: 30
  # Ryzen 9 5900X specific optimizations
  ryzen_9_5900x_optimized: true
  thread_affinity_enabled: true
  # Prioritize performance cores (first 8 cores)
  performance_cores: [0, 1, 2, 3, 4, 5, 6, 7]
  background_cores: [8, 9, 10, 11]
  # Power management settings
  high_performance_plan: true
  disable_core_parking: true
  # AMD-specific optimizations
  precision_boost_overdrive: true
  simultaneous_multithreading: true
```

Memory Configuration

```yaml
memory:
  # Maximum memory usage percentage
  max_percent: 75
  # Threshold for unloading models
  model_unload_threshold: 85
  # XPG D10 DDR4-3200 specific settings
  xpg_d10_memory: true
  # Memory optimization settings
  large_pages_enabled: true
  numa_aware: true
  preload_models: true
  cache_size_mb: 6144
  min_free_gb: 4
  defragmentation_threshold: 70
```

GPU Configuration

```yaml
gpu:
  # Maximum GPU memory usage percentage
  max_percent: 90
```

```yaml
# Threshold for unloading models
model_unload_threshold: 95
# RTX 3080 specific optimizations
rtx_3080_optimized: true
tensor_cores_enabled: true
tensor_precision: "tf32"
# CUDA settings
cuda_compute_type: "float16"
cuda_streams: 3
cuda_memory_pool: true
cuda_graphs: true
# VRAM management settings
max_batch_size: 16
reserved_memory_mb: 256
dynamic_memory: true
fragmentation_threshold: 15
pre_allocation: true
```

Advanced Topics

Custom Hardware Detection

The `HardwareDetector` class can be extended to recognize additional hardware:

```python
from maggie.utils.resource.detector import HardwareDetector

class ExtendedHardwareDetector(HardwareDetector):
    def detect_cpu(self) -> Dict[str, Any]:
        cpu_info = super().detect_cpu()

        # Add detection for additional CPU models
        model_lower = cpu_info['model'].lower()
        if 'intel' in model_lower and 'i9-12900k' in model_lower:
            cpu_info['is_intel_i9_12900k'] = True
            cpu_info['architecture'] = 'Alder Lake'
            cpu_info['recommended_settings'] = {
                'max_threads': 16,
                'affinity_strategy': 'p_cores_first',
                'power_plan': 'high_performance'
            }

        return cpu_info
```

Custom Optimization Profiles

You can create custom optimization profiles for specific workloads:

```python
def create_llm_inference_profile() -> Dict[str, Any]:
    profile = resource_manager.optimizer.create_optimization_profile()

    # Modify the profile for LLM inference
    profile['threading']['max_workers'] = 4
```

```
    profile['gpu']['compute_type'] = 'float16'
    profile['gpu']['tensor_cores'] = True
    profile['gpu']['reserved_memory_mb'] = 512
    profile['memory']['preloading'] = True

    return profile

# Apply the custom profile
def apply_llm_inference_profile():
    profile = create_llm_inference_profile()

    # Apply threading optimizations
    if 'threading' in profile:
        import psutil
        process = psutil.Process()
        if profile['threading'].get('worker_affinity'):
            process.cpu_affinity(profile['threading']['worker_affinity'])

    # Apply GPU optimizations
    if 'gpu' in profile and profile['gpu'].get('tensor_cores', False):
        import torch
        if torch.cuda.is_available():
            torch.backends.cuda.matmul.allow_tf32 = True
            torch.backends.cudnn.allow_tf32 = True
```

Troubleshooting

Common Issues and Solutions

1. **High Memory Usage**

   Issue: The application uses more memory than expected.

   Solution: Adjust memory thresholds in config:

   ```
   memory:
     max_percent: 60  # Reduce from 75%
     model_unload_threshold: 75  # Reduce from 85%
   ```

   And implement manual memory reduction:

   ```
   # Reduce memory usage
   resource_manager.reduce_memory_usage()
   ```

2. **GPU Memory Warnings**

   Issue: Frequent GPU memory warnings during operation.

   Solution: Clear GPU memory before intensive operations:

   ```
   # Before loading models
   resource_manager.clear_gpu_memory()
   ```

Adjust GPU config settings:

```
gpu:
  max_percent: 80  # Reduce from 90%
  reserved_memory_mb: 512  # Increase from 256MB
```

3. **CPU Core Allocation Issues**

   Issue: Thread affinity not working correctly.

   Solution: Verify CPU core allocation:

```
import psutil

# Check current affinity
process = psutil.Process()
current_affinity = process.cpu_affinity()
print(f"Current CPU affinity: {current_affinity}")

# Manually set affinity if needed
process.cpu_affinity([0, 1, 2, 3, 4, 5, 6, 7])
```

Diagnostic Tools

1. **Resource Status Report**

   Generate a detailed resource status report:

```
status = resource_manager.get_resource_status()
print(f"CPU usage: {status['cpu']['percent']}%")
print(f"Memory available: {status['memory']['available_gb']:.2f} GB")
if status['gpu']['available']:
    print(f"GPU memory used: {status['gpu']['memory_percent']:.2f}%")
```

2. **Hardware Report**

   Generate a detailed hardware report:

```
report = resource_manager.get_hardware_report()
print(f"CPU: {report['hardware']['cpu']['model']}")
print(f"Memory: {report['hardware']['memory']['total_gb']:.2f} GB")
if report['hardware']['gpu']['available']:
    print(f"GPU: {report['hardware']['gpu']['name']}")

# Check recommendations
for recommendation in report['recommendations']:
    print(f"Recommendation: {recommendation}")
```

API Reference

ResourceManager

```
class ResourceManager(StateAwareComponent, EventListener):
    """
    Main resource management interface that integrates hardware detection,
    performance optimization, and resource monitoring.
    """

    def __init__(self, config: Dict[str, Any]):
        """
        Initialize the resource manager.

        Parameters
        ----------
        config : Dict[str, Any]
            Configuration dictionary
        """

    def setup_gpu(self) -> None:
        """Set up the GPU environment with optimizations."""

    def start_monitoring(self, interval: float = 5.0) -> bool:
        """
        Start resource monitoring.

        Parameters
        ----------
        interval : float
            Polling interval in seconds

        Returns
        -------
        bool
            Success status
        """

    def stop_monitoring(self) -> bool:
        """
        Stop resource monitoring.

        Returns
        -------
        bool
            Success status
        """

    def clear_gpu_memory(self) -> bool:
        """
        Clear GPU memory cache.

        Returns
        -------
        bool
            Success status
        """

    def reduce_memory_usage(self) -> bool:
        """
        Reduce system memory usage through cache clearing and garbage collection.
```

```
    Returns
    -------
    bool
        Success status
    """

def apply_hardware_specific_optimizations(self) -> Dict[str, Any]:
    """
    Apply hardware-specific optimizations based on detected hardware.

    Returns
    -------
    Dict[str, Any]
        Applied optimizations
    """

def preallocate_for_state(self, state: State) -> bool:
    """
    Preallocate resources for a specific state.

    Parameters
    ----------
    state : State
        Target state

    Returns
    -------
    bool
        Success status
    """
```

HardwareDetector

```
class HardwareDetector:
    """
    Detects and reports on system hardware.
    """

def detect_system(self) -> Dict[str, Any]:
    """
    Detect system hardware.

    Returns
    -------
    Dict[str, Any]
        System hardware information
    """

def detect_cpu(self) -> Dict[str, Any]:
    """
    Detect CPU information.

    Returns
    -------
    Dict[str, Any]
        CPU information
```

```
    """

def detect_memory(self) -> Dict[str, Any]:
    """
    Detect memory information.

    Returns
    -------
    Dict[str, Any]
        Memory information
    """

def detect_gpu(self) -> Dict[str, Any]:
    """
    Detect GPU information.

    Returns
    -------
    Dict[str, Any]
        GPU information
    """

    def get_detailed_hardware_report(self) -> Dict[str, Any]:
        """
        Generate a detailed hardware report with recommendations.

        Returns
        -------
        Dict[str, Any]
            Detailed hardware report
        """
```

HardwareOptimizer

```
class HardwareOptimizer:
    """
    Creates and applies hardware-specific optimizations.
    """

def __init__(self, hardware_info: Dict[str, Any], config: Dict[str, Any]):
    """
    Initialize the hardware optimizer.

    Parameters
    ----------
    hardware_info : Dict[str, Any]
        Hardware information from HardwareDetector
    config : Dict[str, Any]
        Configuration dictionary
    """

def create_optimization_profile(self) -> Dict[str, Any]:
    """
    Create a hardware-specific optimization profile.

    Returns
    -------
```

```
        Dict[str, Any]
            Optimization profile
        """

    def optimize_for_rtx_3080(self) -> Dict[str, Any]:
        """
        Apply RTX 3080 specific optimizations.

        Returns
        -------
        Dict[str, Any]
            Applied optimizations
        """

    def optimize_for_ryzen_9_5900x(self) -> Dict[str, Any]:
        """
        Apply Ryzen 9 5900X specific optimizations.

        Returns
        -------
        Dict[str, Any]
            Applied optimizations
        """

    def optimize_for_current_state(self) -> Dict[str, Any]:
        """
        Apply optimizations based on the current state.

        Returns
        -------
        Dict[str, Any]
            Applied optimizations
        """
```

ResourceMonitor

```
class ResourceMonitor:
    """
    Monitors system resources and triggers warnings when thresholds are exceeded.
    """

    def __init__(self, config: Dict[str, Any], hardware_info: Dict[str, Any],
                 memory_threshold: float = 85.0, gpu_threshold: float = 95.0,
                 event_callback: Optional[Callable] = None):
        """
        Initialize the resource monitor.

        Parameters
        ----------
        config : Dict[str, Any]
            Configuration dictionary
        hardware_info : Dict[str, Any]
            Hardware information from HardwareDetector
        memory_threshold : float
            Memory warning threshold percentage
        gpu_threshold : float
            GPU memory warning threshold percentage
```

```
    event_callback : Optional[Callable]
        Callback function for resource events
    """

def start(self, interval: float = 5.0) -> bool:
    """
    Start resource monitoring.

    Parameters
    ----------
    interval : float
        Polling interval in seconds

    Returns
    -------
    bool
        Success status
    """

def stop(self) -> bool:
    """
    Stop resource monitoring.

    Returns
    -------
    bool
        Success status
    """

def get_current_status(self) -> Dict[str, Any]:
    """
    Get current resource status.

    Returns
    -------
    Dict[str, Any]
        Current resource status
    """

def get_resource_trends(self) -> Dict[str, Any]:
    """
    Get resource usage trends.

    Returns
    -------
    Dict[str, Any]
        Resource usage trends
    """
```

This manual provides a comprehensive guide to using the Maggie AI Assistant resource management package. For additional questions or detailed assistance, please consult the API documentation or contact the development team.