ScheduleLambda audit for thread safety

We take a small chunk of code around ScheduleLambda:

```
include <type_traits>
#include <string.h>
#include <utility>

class LambdaBridge
{
public:
    // Use initialize instead of constructor because this class has to be trivial
    template <typename Lambda>
    void Initialize(const Lambda & lambda)
    {
        // memcpy is used to move the lambda into the event queue, so it must be
trivially copyable
        static_assert(std::is_trivially_copyable<Lambda>::value, "lambda must be
trivially copyable");
        static_assert(sizeof(Lambda) <= 24, "lambda too large");
        static_assert(4 % alignof(Lambda) == 0, "lambda align too large");

        // Implicit cast a capture-less lambda into a raw function pointer.
        mLambdaProxy = [](const LambdaStorage & body) { (*reinterpret_cast<const
Lambda *>(&body))(); };
        ::memcpy(&mLambdaBody, &lambda, sizeof(Lambda));
    }

    void operator()() const { mLambdaProxy(mLambdaBody); }

private:
    using LambdaStorage = std::aligned_storage_t<24, 4>;
    void (*mLambdaProxy)(const LambdaStorage & body);
    LambdaStorage mLambdaBody;
};

int ScheduleLambdaBridge(LambdaBridge && bridge)
{
    return 0;
}

template <typename Lambda>
int ScheduleLambda(const Lambda & lambda)
{
  LambdaBridge bridge;
  bridge.Initialize(lambda);
  return ScheduleLambdaBridge(std::move(bridge));
}
```

And translate it to plain C++ code using cppinsights.io:

```
#include <type_traits>
#include <string.h>
#include <utility>

class LambdaBridge
{

  public:
  template<typename Lambda>
  inline void Initialize(const Lambda & lambda)
  {
```

```
    /* PASSED: static_assert(std::is_trivially_copyable<Lambda>::value, "lambda
must be trivially copyable"); */
    /* PASSED: static_assert(sizeof(Lambda) <= static_cast<unsigned long>(24),
"lambda too large"); */
    /* PASSED: static_assert((static_cast<unsigned long>(4) % alignof(Lambda)) ==
static_cast<unsigned long>(0), "lambda align too large"); */

    class __lambda_18_24
    {
      public:
      inline auto operator()(const LambdaStorage & body) const
      {
        (*reinterpret_cast<const Lambda *>(&body))();
      }

      using retType_18_24 = auto (*)(const LambdaStorage &);
      inline constexpr operator retType_18_24 () const noexcept
      {
        return __invoke;
      };

      private:
      static inline auto __invoke(const LambdaStorage & body)
      {
        return __lambda_18_24{}.operator()(body);
      }


      public:
      // /*constexpr */ __lambda_18_24() = default;

    } __lambda_18_24{};

    this->mLambdaProxy = __lambda_18_24;
    memcpy(&this->mLambdaBody, &lambda, sizeof(Lambda));
  }
  inline void operator()() const
  {
    this->mLambdaProxy(this->mLambdaBody);
  }

  private:
  using LambdaStorage = std::aligned_storage_t<static_cast<unsigned long>(24),
static_cast<unsigned long>(4)>;
  void (*mLambdaProxy)(const LambdaStorage &);
  LambdaStorage mLambdaBody;
  public:
  // inline constexpr LambdaBridge() noexcept = default;
};

int ScheduleLambdaBridge(LambdaBridge && bridge)
{
  return 0;
}

template<typename Lambda>
int ScheduleLambda(const Lambda & lambda)
{
  LambdaBridge bridge = LambdaBridge();
  Initialize(lambda);
  return ScheduleLambdaBridge(std::move(bridge));
}
```

We can see that in the process of scheduling a Lambda block there is no usage of shared data from the stack, therefore the call is thread safe.