

Power Architecture™  
DEVELOPER CONFERENCE '07

A blue car is driving away on a road that curves to the right. The road is flanked by green hills and a body of water. A hand is holding a traffic cone from the left side of the road, and another traffic cone is on the right side. The sky is cloudy.

## Unleashing the Power with Advanced Compiler Optimizations

Tuesday September 25, 2007 15:00-16:00

Greg Davis, Green Hills Software

Power.ORG ™

# Why Compilers?



- » With today's fast CPUs, why do compilers matter?
- » Optimizations:
  - Minimize worst case response time for safety critical applications
  - Faster code:
    - Select less expensive CPU – higher profit margins
    - More CPU “sleep” time – longer battery lifetime
    - More pages per minute, packets per second, etc.
  - Smaller code:
    - Allow more features to be included in product.
    - Longer product lifetime
    - Higher sales price
    - Smaller ROM

# More than just optimizations!



- » Standards compliance
- » Support for new processors
- » Embedded friendly
- » Robust, professionally supported tools

# Standards



- » GHS is fully compatible with C '89/'99.
  - Wide character support from Amendment 1
  - Full C99 support, features in other modes
  
- » GHS is fully compatible with C++, including:
  - STL
  - Exceptions
  - Namespaces
  - Covariance
  - Export templates.
  - Mature, used by customers for years. No caveats, undiscovered incompatibilities, etc.

# Standard Extensions



## » Freescale AltiVec extensions

- New “vector” datatypes
- Intrinsic functions to operate on the data.
- Generic functions to operate on the data
- Uses proper vector initialization syntax

## » Freescale e500 extensions

- Conceptually similar to AltiVec extensions, but syntax is much different.
- Green Hills was first to market with e500 support by over a year.

# ABI Compliance



- » Green Hills has participated in the (E)ABI efforts since the early days at Motorola.
- » Fully compatible with the EABI (embedded) and the Linux ABI .
- » Compatible with the portable assembler standard

# Usage of hardware



- » Compiler allows for full C/C++, even when hardware doesn't directly support certain features
  - Floating point (hardware, software, hardware single, software double, or "double as single" mode)
  - Long longs (uses short instruction sequence on 32-bit machines, uses native instructions on 64-bit processors)
  - Division, square root, etc, is all fully implemented whether the hardware exists or not
- » Aggressive use of hardware when features are available
  - Use of SPE instructions for single precision floating point on e500
- » Emulation routines are hand-coded in assembly for efficiency

# Embedded-friendly compilers



» Not just transplanted UNIX compilers

» Embedded features:

- Interrupt routines in C
- Small data area/zero data area
- Local data area
- Packed data and full `__packed` type qualifier
- Byte reversed type qualifiers
- Special sections, data remapping by linker or compiler
- ROM attributes
- Optimize calls to functions that never return

# Benchmarks



## » Original Dhrystone benchmark

- Too artificial
- Compilers have been tuned for this benchmark

## » Spec2000

- Assumes windows or UNIX environment
- Not reflective of embedded programs

# EEMBC Benchmarks



- » Consortium of silicon and tools vendors
- » Benchmarks are comprised of real world embedded code in different categories
  - Automotive
  - Consumer
  - Digital Entertainment
  - Java
  - Networking
  - Network Storage
  - Office Automation
  - Telecom
- » Works on bare boards – doesn't require a huge operating system for program execution
- » Resistant to compiler tuning
- » Customers believe in benchmarks again

# Green Hills EEMBC Performance



"The Green Hills Software compiler generated the fastest code on the EEMBC Telecommunications suite of benchmarks in our previous certification of the 1-GHz MPC7455 processor. We were pleased to see that the Green Hills compiler provided superior performance on our new AltiVec-enabled, optimized version of this suite as well."

*Chuck Corley, Director Application Engineering, Freescale*



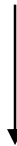
"Green Hills Software's compilers produced the best results on 14 out of 15 of our EEMBC benchmarks."

*Pesh Gala, IBM Strategic Marketing Manager*

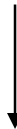
# GHS Compiler Architecture



Front end and language specific optimizations



Global and interprocedural, language and target independent optimizations



Target code generation and optimization



```
func(a*b);  
c = c + a*b;
```

Front end

```
temp = a*b;  
func(temp);  
c = c + temp;
```

Global optimizer

```
mullw r31, r3, r4  
mr r3, r31  
bl func  
add r30, r30, r31
```

PowerPC machine code generator

# Optimizations



» At this point, we will look into a few of the interesting and unique optimizations that Green Hills uses to generate the industry's best code.

# Common Subexpression Elimination (CSE)



- » A classic optimization, well known for many years in the research community
- » It takes computations that are redundant (known to have the same value) across all paths and saves the result after the first computation, so that it need only be computed once

# CSE Example



Original code:

```
foo(a * b);  
if (a > 0)  
    c = a * b;  
else  
    d = a * b;
```

Optimized (saves 1 multiply)

```
temp = a * b;  
foo(temp);  
if (a > 0)  
    c = temp;  
else  
    d = temp;
```

# Partial Redundancy Elimination (PRE)



- » A newer algorithm. Developed in 1979 by Morel, but not very useable. Reformulated in 1994 by Knoop and later by others
- » More aggressive than CSE. Does not require that a computation is redundant across all paths, but just that it is redundant across some.
- » Handles loops in the control flow and partial invalidations

# PRE Example



Original code:

```
extern int a, b;  
loc = a*b;  
if (loc < 0)  
    recompute();  
return a*b;
```

Optimized code:

```
extern int a,b;  
loc = tmp = a*b;  
if (loc < 0) {  
    recompute();  
    tmp = a*b;  
}  
return tmp;
```

# PRE versus CSE



Original code:

```
extern int a, b;  
loc = a*b;  
if (loc < 0)  
    recompute();  
else if (loc == 0)  
    recompute2();  
return a*b;
```

Optimized code:

```
extern int a, b;  
loc = tmp = a*b;  
if (loc < 0) {  
    recompute(); tmp = a*b;  
} else if (loc == 0) {  
    recompute2(); tmp =  
    a*b;  
}  
return tmp;
```

## PRE/CSE benefits



- » We have found that in addition to the obvious places where the code is redundant, PRE/CSE also helps in a lot of address calculations

```
arr[i] + arr[i+3]
```

→

```
ptr = arr + i; ptr[0] + ptr[3];
```

- » Have added our own ideas to the original optimization.
- » Significant code size reduction (4%-10%) across an entire application

# AltiVec Vectorization



- » Increasingly, new CPUs have the capability of performing multiple computations at once on different pieces of data. This is called Single Instruction Multiple Data (SIMD)
- » Case in point: PowerPC “AltiVec” processors can do 4 floating point operations in parallel with one instruction

# Manual Vectorization



- » Can get very good performance improvement
- » Downsides:
  - User needs to figure out how to get parallelism
  - Needs to learn a number of new datatypes and intrinsics
  - Needs to worry about alignment restrictions

# Automatic Vectorization



```
sum = 0.0F;  
for (i = 0; i < N; i++)  
    sum += x[i] * y[i];
```

Autovectorize. 400% speedup!!

↓

```
for (i = 0; i < N/4; i++)  
    acc = vec_madd(acc, x[i], y[i]);  
accv = acc; /* Recombine */  
sum = accf[0] + accf[1] + accf[2]  
      + accf[3];
```

# Advanced Instruction Selection



- » Green Hills uses an aggressive form of instruction selection to find the fewest instructions to perform a computation.

# Advanced Instruction Selection



```
extern unsigned char arr[];
int foo(int val, int offs, int
    limit)
{
    int sum = val + offs;
    if (sum == 0)
        return 0;
    sum += arr[0];
    if (sum > limit)
        sum = 0;
    return func(sum, arr);
}
```

```
foo:
    add. r3, r3, r4
    beqlr
    lis    r4, %hiadj(arr)
    lbzu  r12, %lo(arr)(r4)
    add   r3, r3, r12
    cmpw r3, r5
    iselgt r3, r0, r3
    b     func
```

# Constant Propagation



- » This is a very old and well known optimization
- » The basic idea is to look ahead from a place where a constant is assigned to a variable and replace all uses of that variable with the constant
- » But a good constant propagator does more...

# Constant Propagation Example



Original code:

```
int foo()
{
    int var = 3;
    return var;
}
```

Optimized code:

```
int foo()
{
    int var = 3;
    return 3;
}
```

# Constant Propagation Example



Original code:

```
int foo(int a, int b)
{
    int var;
    if (a > 0)
        var = 3;
    return (b > 0) ? var : 0;
}
```

New code: (tricky because var is not always written)

```
int foo(int a, int b)
{
    int var;
    if (a > 0)
        var = 3;
    return (b > 0) ? 3 : 0;
}
```

# Constant Propagation Example



```
int glob;
int constant(int x, int y)
{
    int loc = x;
    if (y)
        loc = 3;
    if (loc == 3)
        glob++;
}
```

```
int glob;
int constant(int x, int y)
{
    int loc = x;
    if (y) {
        loc = 3;
        goto lab;
    }
    if (loc == 3) {
lab:
        glob++;
    }
}
```

# Alpha and Omega Motion



- » Code size optimization that attempts to move similar instructions from multiple blocks into a single block
- » Sometimes called “busy code motion” in the literature.
- » “Alpha” refers to moving code upward. “Omega” means the code is moving downward

# Omega Motion Example



Original code:

```
void foo(int i)
{
    if (i)
        glob = 1;
    else
        glob2 = 1;
}
```

New code:

```
void foo(int i)
{
    int *ptr;
    if (i)
        ptr = &glob;
    else
        ptr = &glob2;
    *ptr = 1;
}
```

# Whole Program Optimization



- » The optimizations I have shown up to now are mostly done on individual functions.
- » Much more can be accomplished by optimizing between functions.

# Inlining



» Inlining, a very common optimization, replaces a call with the body of the called function

```
int foo(int i)
{ return i+1; }
```

```
int foo(int i)
{ return i+1; }
```

```
int func(int i)
{ return foo(i)*3; } →
```

```
int func(int i)
{ return (i+1)*3; }
```

# Intermodule Inlining



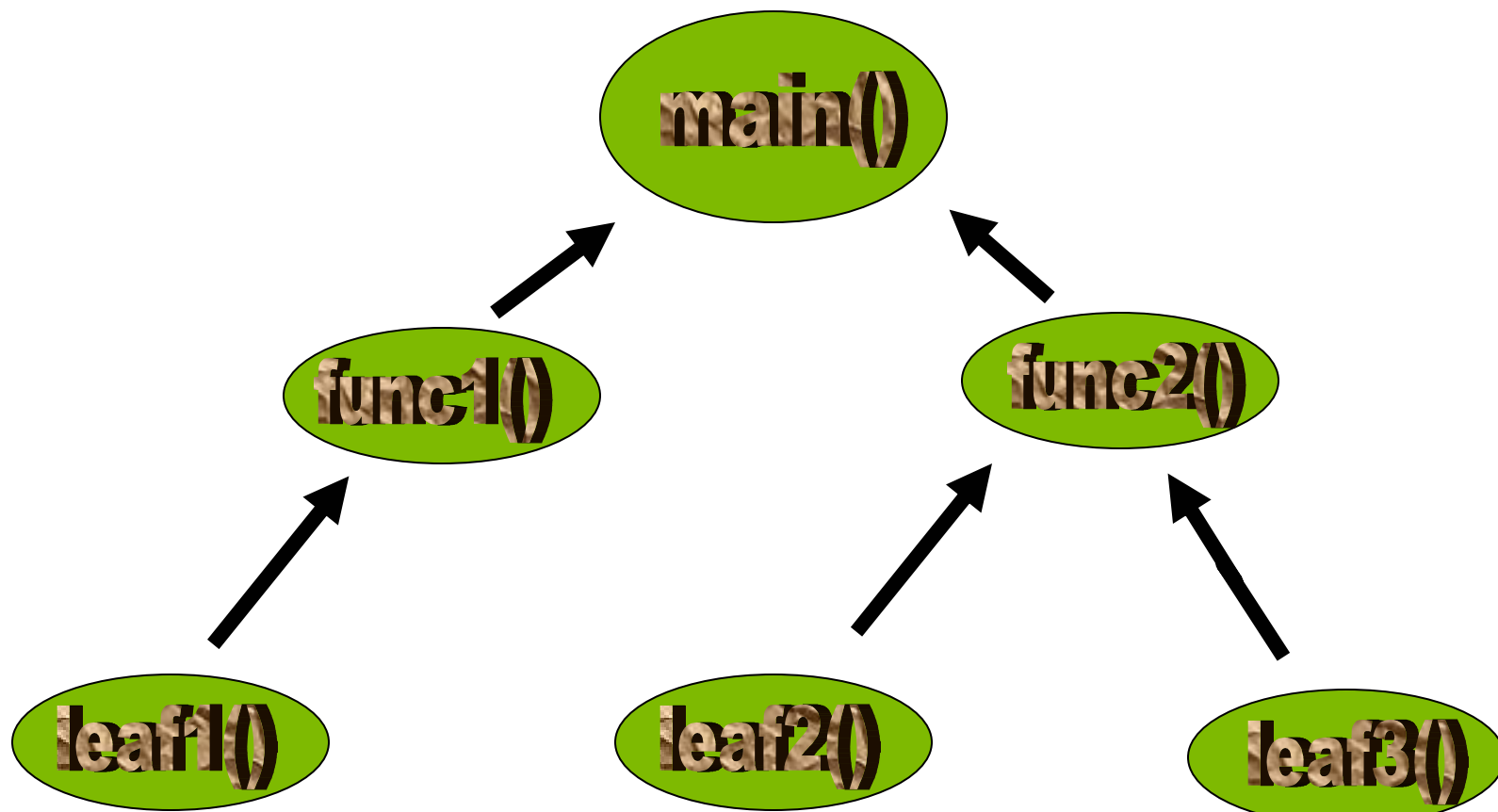
- » Green Hills extends the idea of inlining to work across files. Large projects defeat the inlining capabilities of traditional compilers since a traditional compiler can only see one file at a time
- » The Green Hills compiler gets around this by using an intermodule database for functions
- » This same database can be used for other intermodule optimizations...

# Interprocedural Optimizations



- » Traditional compilers perform only a few interprocedural optimizations:
  - Inlining
  - Allocation into volatile registers across safe calls
  - Local data area allocation
  
- » Powerful host computers are capable of analyzing most embedded applications in their entirety.
  
- » Two modes: Whole program and interprocedural.

# Unidirectional Interprocedural Information Flow



# Unidirectional Information Flow Example



## Original code:

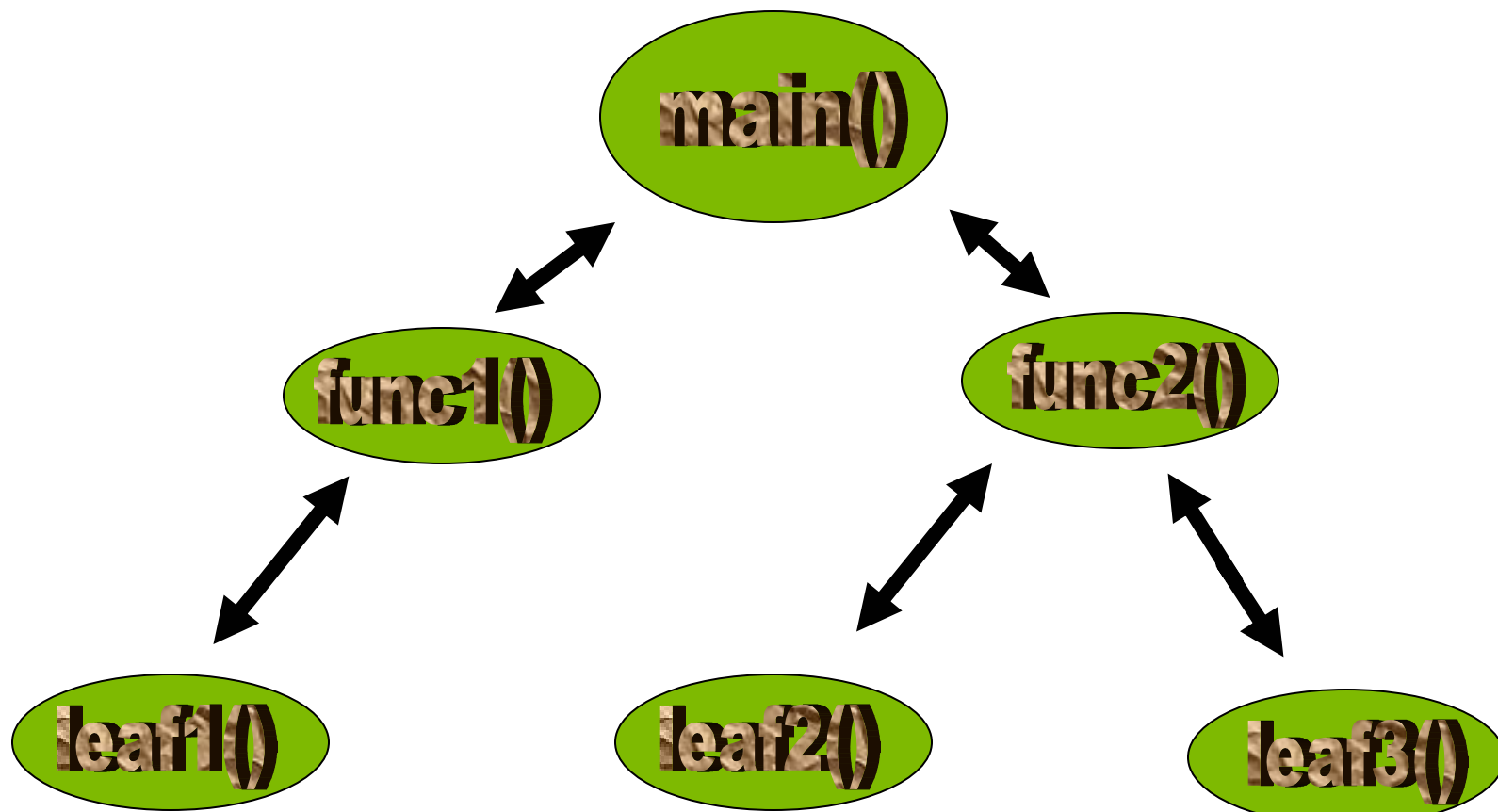
```
int glob;  
int arr[2];  
void record(int a, int b)  
{  
    arr[0] ^= a;  
    arr[1] ^= b;  
}  
  
int func(int val)  
{  
    glob = 0;  
    record(val, 0x8000);  
    return glob;  
}
```

## New code:

```
int glob;  
int arr[2];  
void record(int a, int b)  
{  
    arr[0] ^= a;  
    arr[1] ^= b;  
}  
  
int func(int val)  
{  
    glob = 0;  
    record(val, 0x8000);  
    return 0;  
}
```



# Bidirectional Whole-Program Information Flow



# Bidirectional Information Flow Example



```
int glob;
int can_be_simple(int x, int y)
{
    if (x != 0 || y > 2)
        do_large_computation();
    glob += y;
    return x+y;
}
int is_simple()
{
    return can_be_simple(0, 1) +
           can_be_simple(0, 2);
}
```

```
int glob;
int can_be_simple(int y)
{
    glob += y;
    return y;
}
int is_simple()
{
    return can_be_simple(1) +
           can_be_simple(2);
}
```

# CodeFactor



- » Reduces code size at link time by finding redundant code sequences and pulling them out into pseudo-functions. Example:

Original code:

```
...  
x = arr[0] + arr2[0];  
...  
y = arr[0] + arr2[0];
```

Factored code:

```
...  
x = func();  
...  
y = func();  
func() { return arr[0] +  
arr2[0]; }
```

# Challenges



- » Implementing optimizations is difficult
- » Academic optimizations don't work "as is" in the real world
- » One has to be careful when implementing advanced compiler optimizations so that reliability and stability throughout the product line is not compromised, especially for safety-critical applications
- » Modular compiler structure allows optimizations to be shared between different processors. Green Hills realizes economies of scale.

# Conclusion



- » Green Hills compiler's are one of many advantages that Green Hills Software has over the competition
  
- » Green Hills compilers offer superior:
  - Compliance to international language standards and industry extensions
  - Support for new processors
  - Embedded-oriented compilers
  - Professional support

**Power Architecture™**  
DEVELOPER CONFERENCE

**'07**



# » Subtitle Page