

**OPTIMIZING SPEED VS. SIZE**  
**USING THE CODEBALANCE UTILITY**  
**FOR ARM/THUMB AND MIPS16 ARCHITECTURES**

---

**WHITE PAPER**  
**NOVEMBER 1998**

## INTRODUCTION

---

Not all parts of a microprocessor program require high-speed performance. In some applications, a more important aspect is the size of the code in RAM or ROM. It is this requirement that dual-purpose architectures such as ARM/Thumb and 16-bits were designed to address. Thumb and MIPS16 are architecture extensions whereby an otherwise 32-bit architecture (ARM and MIPS) is enabled to process 16-bit instructions. The use of 16-bit instructions would double available RAM or ROM space if each instruction could do the same work as a 32-bit instruction. This is not always the case. In general, a 16-bit instruction is limited in the number of Op Codes it can call upon and in the number of registers and addressing modes it can reference. Thus, while each instruction can still perform a 32-bit operation, the limitations on registers, addressing and Op Codes makes the 16-bit instructions less powerful than 32-bit instructions.

Therefore, more of them are needed to do equivalent work. Still, the net result is a reduction in total code size with a lesser reduction in performance. By utilizing an embedded compact code microprocessor, designers can cut memory cost by up to 40 percent, or they have the option of using that extra memory space to support additional features. In MIPS16 and ARM/Thumb, code density is improved since there are fewer operands and registers and, consequently, smaller (16-bit) instructions resulting in reduced code size and memory requirements.

Individual code modules can be recompiled at the programmer's choice to use either 32-bit or 16-bit instructions. He or she can write in assembly code or high level languages such as C, C++, or the new Embedded C++ (EC++), and then at compilation time select between 32- or 16-bit code. When all the different code modules are combined into a single binary for a given application, the programmer has the opportunity and flexibility to mix and match 32- and 16-bit code. Some modules can be compiled to 32-bit ISA, others to 16, and all can be gathered together into a single binary. To execute such an application efficiently, the processor must support a low-overhead (single-cycle) method of switching between 16-bit instruction mode and 32-bit instruction mode, or it must handle both types of instructions as encountered.

Some types of functions with high instruction cache-miss rates could realize the effect of doubling the effective cache size through the smaller instruction size. This effect is likely to yield a small improvement in performance. But as far as access to off-chip memory, nearly a full twofold speedup of instruction fetches is achieved as a result of two instructions being fetched on each cycle, which improves the effective bus access speed. This makes sense for large programs that don't fit in cache and are infrequently used.

## BEST OF BOTH WORLDS

---

Code compiled for ARM/Thumb or MIPS16 is typically much smaller than the same code compiled for ARM or MIPS-32 at the expense of slower execution. But it costs nothing to switch between 16-bit and 32-bit modes during execution. Therefore, the natural and effective way to take advantage of these dual-instruction-size architectures is to build the embedded system so that time-critical routines are compiled as 32-bit instructions, while less critical routines are compiled as 16-bit instructions. This general strategy enables embedded applications to achieve a much smaller footprint at relatively small performance expense. Hence, code is Slim and Fast, and you get the best of both worlds.

But how do you do that? First of all, unless the programmer is willing to put each and every individual function into its own separate source code file, the compiler must be able to selectively produce 16-bit or 32-bit instructions on a function by function basis. Still, how does the programmer know which routines to compile as 32-bits and which to compile as 16-bits? The programmer may know enough about the application to "eyeball" it. Time-critical routines

can be built with 32-bit instructions, and less executed routines can be made 16-bits by surrounding the functions with #pragma constructs in the source code (or corresponding directives in hand-written assembly code) or by grouping these functions into their own source modules and compiling in a given mode.

This process is not precise and may be difficult with more complex applications involving hundreds of functions. Guesstimates as to which routines are time critical may not match the true run-time characteristics of the application as exhibited from a run-time execution profile. Similarly, the person writing the code may not be the same as the person who is building it to run within the embedded system or optimizing it to run efficiently.

## OPTIMIZE FOR SMALLER AND FASTER CODE

An efficient design approach is to automate this process utilizing a utility like Green Hills' "CodeBalance." This tool examines code size of the application as built for 32-bits versus 16-bits and a run-time execution profile of the 32-bit application to determine the estimated size/speed tradeoffs for the application on a per function basis. Also, the programmer receives a set of command line options that are output by this utility based on user-supplied constraints. These options can be subsequently passed to the compiler which divides source code into 16-bit and 32-bit components. This precludes the need to change source code with #pragma or to rearrange the functions within their source files.

CodeBalance uses a Graphical User Interface (Fig. 1) to allow the user to specify various options for the optimization process. With a few points and clicks, CodeBalance produces a table of information to guide the user to the next phase of the speed/size tradeoff.

CodeBalance produces a table like that shown in Fig. 2. A list of routines common to both programs is shown under the heading, "Functions". Since the purpose of CodeBalance is to determine speed/size tradeoffs, routines not common to both programs, or routines which have the same exact code size for both programs are not displayed in Fig. 2.

The "fastprog" column shows code size in bytes for each routine built for 32-bits. The next two columns, "msec" and "cum%", display the profile timing information for "fastprog." Column "msec" shows the time spent in each routine listed; the "cum%" column shows the cumulative percentage of runtime, starting from the most time-critical or bottleneck routines. Column "smallprog" then shows the code size in bytes of "smallprog."

The last column "Code-Savings" shows the cumulative estimated potential code size savings resulting from compiling the least time-critical routines in 16-bits mode. For example, if the functions *main()* and *polarize()* are compiled for 16-bits and the remaining functions are left as 32-bits, the estimated code savings is 13 percent. By adding the

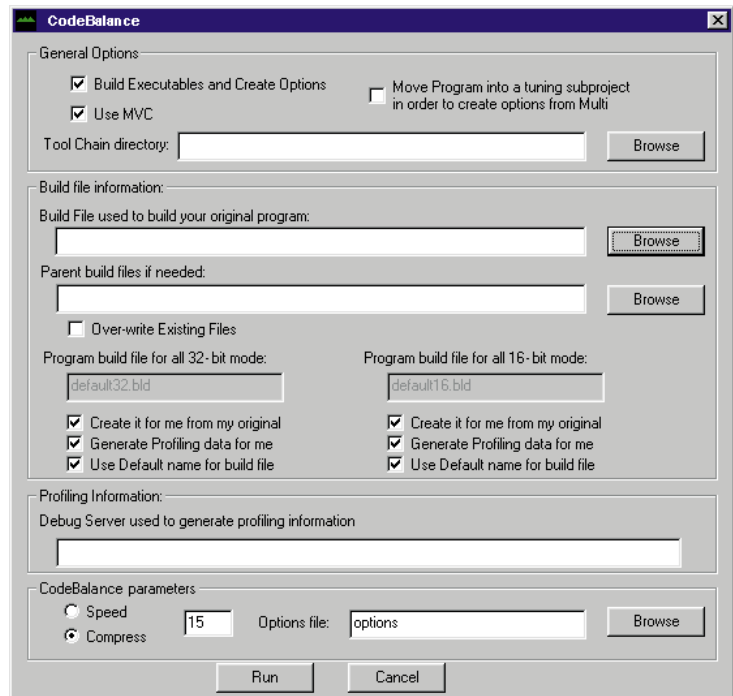


Fig 1: Easy-to-use interface allows the user to specify various options for the optimization process.

function *eval\_parabolic()* to the list of 16-bit compiled functions, the savings increases to 22 percent.

The user can select a code compression goal as a percentage of overall code size. For example, by selecting a code compression goal of 35%,

CodeBalance will recompile routines in 16-bit mode until a 35% reduction in code size is achieved. Then, CodeBalance will re-profile the application to determine the impact on performance and report this data to the user.

Function:	fastprog:	msec:	cum%:	smallprog:	Code Savings:
gen_pairs	196	4.788	45	150	45%
calc_sum	272	1.792	62	186	42%
ion_propagate	376	1.752	79	140	37%
eval_parabolic	232	0.717	86	94	22%
main	244	0.130	87	54	13%
polarize	8	0.080	88	4	0%
<b>program total</b>	<b>1544</b>	<b>10.533</b>		<b>844</b>	<b>45%</b>

Fig. 2 Initial CodeBalance analysis of program.

Another type of tuning option enables the user to specify the percentage of execution time to be allocated to 32-bit routines. Here, CodeBalance chooses the most time-critical routines such that <percent> of the total execution time is compiled for 32-bits. This method ensures that bottleneck routines are kept as fast as possible. Leftover routines are compiled for 16-bits to save space. For example, to ensure that at least the top 75 percent of the bottleneck code is compiled for maximum speed, CodeBalance would compile *gen\_pairs*, *calc\_sum*, and *ion\_propagate* in 32-bit mode for maximum speed, and the rest in 16-bit mode for minimum code size.

CodeBalance can generate an execution profile for the composite application “*composite*” so that the size/speed tradeoff between the original all 32-bits application and the new improved program can be seen (Fig. 3). Here, CodeBalance is shown to have produced a very large code compression of 37 percent with only a 4% loss in performance. (The code sizes 1544 and 976 in Fig. 3 are not the total code sizes for the two executables. Rather, they are the total sizes for the functions that are common between “fastprog” and “composite”. Also, this example is able to use 16-

bit mode extensively. So the 32-bit function prologues and epilogues that are normally present are not linked in.)

Function:	fastprog:	msec:	cum%:	composite:	msec:	Code Savings:
ion_propagate	376	1.752	17	140	2.269	37%
eval_parabolic	232	0.717	23	94	0.916	22%
main	244	0.130	25	4	0.200	13%
polarize	8	0.080	25	4	0.080	0%
<b>program total</b>	<b>1544</b>	<b>10.533</b>		<b>976</b>	<b>10.954</b>	<b>37%</b>

Fig. 3 CodeBalance analysis of size and performance of composite program.

## CONCLUSION

A utility like CodeBalance is of considerable value in embedded designs because it locates the best candidates for 16-bits compilation and, therefore, unlocks the true power of the ISA. Tuning options passed to this utility can be varied to find the desired tradeoff of speed versus size in a composite 32-bits/16-bits application.

## **GREEN HILLS SOFTWARE, INC.**

### **CORPORATE HEADQUARTERS**

30 West Sola Street  
Santa Barbara, California 93101  
Phone: 805.965.6044  
Fax: 805.965.6343  
Email: sales@ghs.com  
URL: www.ghs.com

### **NORTH AMERICA**

California - Cupertino  
T: 408.873.4930 ■ F: 408.873.4933

California - San Clemente  
T: 949.369.3950 ■ F: 949.369.3959

California - Santa Barbara  
T: 805.965.6044 ■ F: 805.965.6343

California - Scotts Valley  
T: 408.430.0525 ■ F: 408.430.0415

Colorado - Denver  
T: 303.740.8462 ■ F: 303.740.8468

Illinois - Chicago  
T: 312.946.5460 ■ F: 312.946.5462

Massachusetts - Lexington  
T: 781.862.2002 ■ F: 781.863.2633

North Carolina - Raleigh  
T: 919.846.7340 ■ F: 919.676.7005

Pennsylvania - King of Prussia  
T: 610.768.7756 ■ F: 610.768.7781

Texas - Dallas  
T: 972.733.6505 ■ F: 972.733.6504

### **NORTH AMERICAN ADA SALES**

California - Laguna Hills  
T: 949.460.6442 ■ F: 949.460.6443

Florida - Palm Harbor  
T: 813.781.4909 ■ F: 813.781.3915

### **INTERNATIONAL OFFICES**

#### **EUROPEAN HEADQUARTERS**

United Kingdom - London  
T: +44.1.494.429.336 ■ F: +44.1.494.429.339

Germany - Munich  
T: +49.7.21.986.2580 ■ F: +49.7.21.986.2581

France - Paris  
T: +33.1.46.96.07.00 ■ F: +33.1.46.96.07.07

Netherlands - Amsterdam  
T: +31.33.433.0827 ■ F: +31.33.433.0828

