

A Post-Compilation Register Reassignment Technique for Improving Hamming Distance Code Compression

Montserrat Ros¹

School of Mathematics, Statistics
and Computer Science
The University of New England
Armidale, Australia
+61 2 6773 2286

ros@mcs.une.edu.au

Peter Sutton

School of Information Technology
and Electrical Engineering
The University of Queensland
Brisbane, Australia
+61 7 3365 4854

p.sutton@itee.uq.edu.au

ABSTRACT

Code compression is a field where compression ratios between compiler-generated code and subsequent compressed code are highly dependent on decisions made at compile time. Most optimizations employed by compilers tend to focus on parameters such as program performance, minimizing resource dependencies and sometimes the option of reducing code size.

This paper describes a post-Compilation technique for the greedy reassignment of general purpose scratch registers to improve Hamming distance based code compression. The code translation renumbers registers based on the frequency of registers used by isomorphic instructions and employs a Gray coding scheme to reduce Hamming distances between similar instructions.

Register reassignment has been successfully implemented in areas where the compiler optimizations do not include a particular metric, for example, power savings. Program values can be reassigned register numbers that reduce overall power consumption of the address bus and register file decoder, at no cost to code size or performance.

The application of the register reassignment technique in this paper reduced the number of dictionary vectors required by a program on average by 9.74%. Code compression ratios of register-reassigned binaries were consistently around 3-4% (of original program size) lower than code compression applied to original binaries, with the highest such reduction at nearly 7%. General purpose register reassignment is a technique that allows for gains in compression efficiency with no penalty in hardware. Other techniques that could be trialed include commutative register switching, dead register detection and assignment and complete register re-allocation.

Categories and Subject Descriptors

D.3.4 [Programming Languages]: Processors – *code generation, Compilers, Optimization.*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

CASES'05, September 24–27, 2005, San Francisco, California, USA.
Copyright 2005 ACM 1-59593-149-X/05/0009...\$5.00.

General Terms

Algorithms, Measurement, Experimentation.

Keywords

Code Compression, Hamming Distance, Register Reassignment.

1. INTRODUCTION

With the increased use of microprocessors in embedded applications, the need for smaller, more power efficient and cost effective processors has promoted research into improving these areas. Whilst sub-micron technologies are continually advancing, there are other areas which can be inspected for the improvement of performance.

Code size management is one of a number of significant issues for embedded system designers. One solution, the compression of instructions residing in memory, has received attention in recent years for its ability to provide either chip size reduction for on-chip solutions or larger effective memory spaces. As consumers require more functionality from their embedded devices, the area of code size management will continue to be of importance.

Successful code compression architectures have been incorporated in several processors to date. The IBM PowerPC 405 uses the CodePack scheme, a piece-wise Huffman scheme where the most frequent symbols are assigned smaller codewords [7]. The Atmel Diopsis also implements a compressed code architecture whereby 128-bit instruction words are compressed to an average of 50 bits per instruction word, highlighting the advantage of integrated instruction set architecture (ISA) and code compression architecture [3].

Code compression efficiency is widely defined [4, 10, 13, 18] as the ratio between the compressed program size and the original program size. This is the main metric we will use to determine if the register reassignment has improved compression.

$$\text{compression ratio} = \frac{\text{compressed program size}}{\text{original program size}}$$

¹ This work was carried out in the author's capacity as a student of The University of Queensland.

Clearly, the smaller the compression ratio, the better the compression. The compression ratio depends on the size of the original compiler output. Our previous work has found that the smallest overall sizes after compression are obtained when the smallest possible compiler build is used [19].

In this paper, we present a register reassignment scheme based on register-pair frequencies for the purpose of improving Hamming distance based code compression. Section 2 presents background and related work in this field. Section 3 describes the implementation details of the reassignment technique and compression applied. Section 4 reports results from applying the compression to binaries before and after both Gray code and register-pair frequency reassignment and Section 5 concludes with a discussion and comparison of results.

2. BACKGROUND AND RELATED WORK

Many modern compilers implement a version of graph-coloring for the purpose of allocating program values to registers. Generally, there are many more program values than the number of available registers, however, not all program values are ‘live’ at all points of execution in the program. Essentially, this algorithm is based on identifying pairs of program values that, at some point in the program, must both be stored in registers (i.e. are both ‘live’). These program values can therefore not be allocated to the same register, and are considered to *interfere*. The graph representing all such interactions between program values is often termed an *interference graph*, and the solution to an N -coloring of this graph is equivalent to a solution of the program’s register allocation to N registers. If an N -coloring cannot be found, extra instructions are added to store and load values from memory when needed. This is called ‘spilling’ a program value into memory. The problem of finding an N -coloring is an NP-complete problem. Muchnick [15] contains further information and a complete explanation of this algorithm.

Register allocation is the process of determining which program values should be placed in registers at each point in time in the execution of a program while *register assignment* is the process of determining in which register the value will be stored. In general, works regarding the reassignment or reallocation of registers concentrate on general purpose scratch-pad registers that are usually assumed to contain nothing and can be left in any state. In general, these registers are interchangeable. Special purpose registers are not included in these studies, as this would lead to changes in the architecture.

2.1 Code Compression Related Work

The area of text or data compression is a mature one, but code compression dates from 1992, when Wolfe and Channin published a paper on a Compressed Code RISC Processor (CCRP) [22]. The CCRP involved the compression of code and a ‘code-expanding instruction cache’, such that the decompression could be transparent to the processor.

Research in this field has mostly concentrated on code compression systems that are software-based or where hardware need only be altered slightly in order to achieve a saving of program size. An example of the latter situation would be the inclusion of a decompression engine next to a processor core in an ASIC embedded design.

In the RISC literature, Lefurgy et al presented dictionary compression in [11] where all unique instructions are recorded in an ‘instruction table’ and each instruction is replaced by an index into the table. Liao et al offered a dictionary compression scheme based on set-covering in [14] which looks at substrings that occur frequently. Lekatsas presented a semi-adaptive dictionary compression scheme in [13] which generated new opcodes for instructions appearing frequently. Some software/compiler methods have also been presented in [5, 6, 12].

In the VLIW literature, Ishiura and Yamaguchi [8] investigated code compression based on Automatic Field Partitioning. Larin and Conte [9] compared code compression methods and a tailored encoding of the Instruction Set Architecture. Xie et al. [23] used a reduced-precision arithmetic coding technique combined with a Markov model and also present a Tunstall-based memory-less variable-to-fixed encoding scheme with improved Markov variable-to-fixed algorithm in [24].

Nam et al [16] presented a dictionary compression method and compared the performance of “identical” and “isomorphic” instruction word compression schemes. Isomorphic instruction words were defined as words that contain the same set of opcodes, but the operands used are different; or words that contain the same set of operands with different opcodes. We will use the term ‘*isomorphic*’ as a relationship between two instructions where all non-register fields are identical (ie opcodes, conditional bits, etc – only the register operands may differ).

Prakash et al [18] present a dictionary based encoding scheme that divides instructions into two 16-bit halves. For each half, a dictionary is constructed that contains a choice set of vectors such that a majority of the vectors used throughout the program in that half of the instruction differ from one of the dictionary vectors by a Hamming distance of at most one. The Hamming distance between two binary vectors is defined as the number of bit locations where bits differ.

In a previous paper [20], we presented an implementation of Hamming distance based code compression which selects a reduced dictionary such that every program vector differs by a Hamming distance of at most three. That code compression scheme is applied in this paper and is described in more detail in Section 3.4.

2.2 Register Reassignment Related Work

A recent paper on a compiler-driven register name adjustment (RNA) is very similar in concept to our work, though with very different objectives and methods. In [17], Petrov and Orailoglu describe a register permutation algorithm that uses the frequencies of register pairs (groups of two registers that occupy the same field in adjacently executed instructions) to permute the register indices so as to reduce the Hamming distance between frequent register pairs. The motivation for this is that the number of bit transitions in the streams of register fields is a major factor in the power consumption on the address bus and the register file decoder. The more bit transitions between adjacent instructions, the more power dissipated. Power reductions of 50% to 60% are achieved across several applications.

Petrov and Orailoglu construct a register histogram graph, with registers as vertices and each positively weighted edge corresponds to the frequency of that register pair. This is

essentially the same type of information as conveyed in our register-pair frequency matrix, except that we are not interested in register pairs corresponding to the same field of two adjacently executed instructions. Instead, our focus is regarding register pairs in the same field of two isomorphic instructions, and we use the Gray code sequence to assist us in the selection of a chain of register numbers that are guaranteed to be one Hamming distance from their neighbors.

In [25], Zhang et al. describe a post-pass register re-allocation scheme for binary translation of compiler output for the purpose of reducing overall energy consumption. They target the dynamic activities in the processor cache and achieve power savings by reducing the number of register spills which result in further load/store instructions for maintaining that program value outside of the register file. They use the fact that many register allocators still leave dead/unused registers where spills exist. Their incremental solution takes the compiled binary and considers dead/unused registers, hot regions and spills identification and performs a costs analysis before inserting compensation code. Overall, dynamic spill load/stores were reduced by 0% - 26.4%.

3. IMPLEMENTATION

Like [17], we too wish to exploit the fact that there is no pre-ordained correspondence between program variables and register names. Because general purpose registers are interchangeable, a new register permutation can be devised to minimize Hamming distance. This means that when the Hamming distance based compression scheme is applied to the new binary, fewer dictionary entries and bit toggling are required for the compression.

Three approaches have been considered to solve this problem:

1. reassignment (or renaming) of the registers with a static sequence for all programs (which assumes a permutation can be chosen that minimizes overall Hamming distances between instruction register fields);
2. reassignment (or renaming) of the registers with a sequence dependent on the program characteristics (which register pairs should be the smallest Hamming distances apart); and
3. reallocation of the registers, incorporating Hamming distance into the optimization parameters for the compiler.

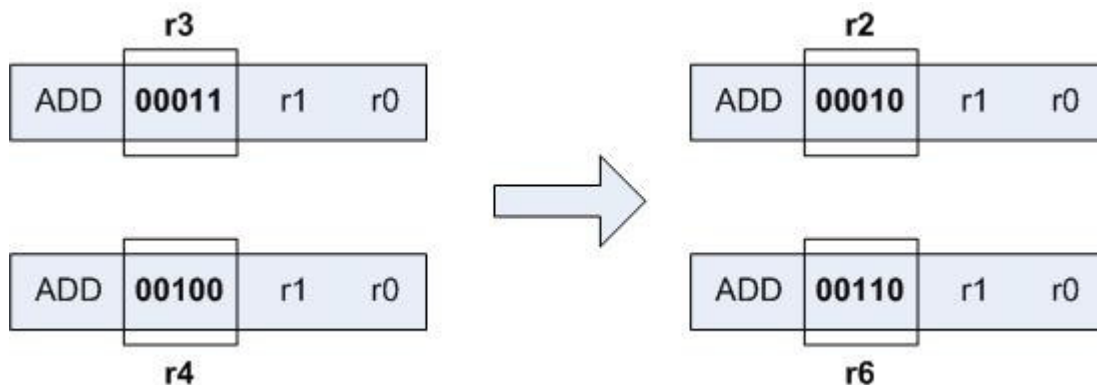


Figure 1 – Example of Gray Code Register Reassignment

The first and second options leave the register structure as they have been allocated at compile time by the compiler, but simply rename the registers. These are the methods described in this paper. The third option changes the structure of registers throughout the program and is much more complex than the first two, as it involves taking into account the many other facets a compiler is required to optimize. In [17] the comment on option 3 is:

“... revisiting the problem of register allocation is impractical because compiler technologies attempt optimizations that aim at satisfying multiple objectives. Nevertheless, we can still use a permutation of the register indices while retaining all the compiler’s previous optimizations.”

3.1 Gray Code Register Reassignment

Perhaps the simplest register reassignment scheme would be to assign each register with its Gray code encoding. This would mean that isomorphic instructions that accessed r3 (00011) and r4 (00100) in the same field would be reassigned to the Gray code representation of r3 (00010 = r2) and r4 (00110 = r6), reducing the hamming distance from 3 to 1. This example is shown in Figure 1. If this register pair constituted a very common register pair between isomorphic instructions, this could mean a big saving for the purposes of Hamming distance based compression.

Using a static Gray code sequence makes the assumption that adjacent registers in the register file (eg r3 and r4) would be accessed commonly by isomorphic instructions and would require a Hamming distance of 1 between them – which may not be the case.

3.2 Register-Pair Frequency Reassignment

A more fitting solution would be to generate a dynamic sequence of register indices where adjacent registers do correspond to pairs of common registers (within isomorphic classes for a particular program), and then apply the Gray code sequence to those register indices.

The register-pair frequency reassignment algorithm is described below.

Step 1: Partition Instructions into Isomorphic Classes

Read in the instructions from the binary file and partition them

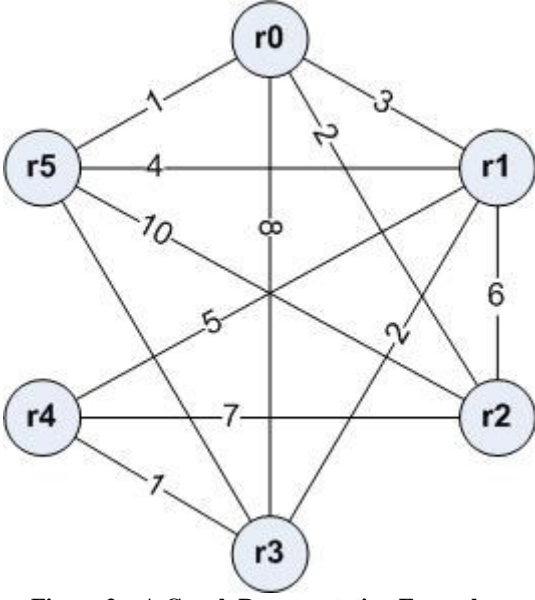


Figure 2 – A Graph Representation Example of Weighted Register Pairs

into classes of instructions that are isomorphic. As mentioned in section 2.1, for this work we define isomorphic instructions as instructions which are identical except for their register operands.

Step 2: Generate register-pair frequency matrix

This step is analogous to creating a graph with registers as vertices, and weighted edges corresponding to the number of times the two registers are used in the same field in isomorphic instructions. An example of such a graph is given in Figure 2.

This is implemented as a register-pair frequency matrix, where the frequency of a register pair is recorded at the location referenced by the registers as indices. We generate the register-pair frequency matrix by comparing every pair of isomorphic instructions in the program and recording the registers that differ. This involves, within each isomorphic class containing at least two instructions, taking each pair of instructions and noting the register operands that differ, as register pairs. These pairs are used as an index pair and the corresponding count in the matrix is incremented. Thus, if two instructions are isomorphic except for one register field, where the first instruction contains register rA and the second instruction contains register rB , then the corresponding matrix index rA, rB is incremented.

As an entire class of instructions can be processed together, a simple tally of the frequency of each register in a given field can be used to increment the register-pair frequency matrix using the following formula:

$$RPFM[rA][rB] += count(rA) \times count(rB)$$

Step 3: Construct Register Chain

This step takes the graph from step 2 and constructs a new graph which will contain a chain made up of edges corresponding to register pairs that must be encoded a Hamming distance of 1 apart. Thus, the new graph initially contains just the vertices from the graph from step 2 and edges are added to this graph in order of frequency (most weighted edges first), even if the edge is disjoint.

If the addition of an edge would result in the creation of a cycle, or would create a node with more than two edges, this is not allowed, and the next edge is reviewed for addition to the graph. Figure 3 shows the same graph as Figure 2, with the chain edges highlighted. Edges are added one by one in order of frequency and the notation $(ra, rb) \Rightarrow (ra, rb, rc)$ indicates that after adding (or attempting to add) register pair (ra, rb) , the chain is updated and looks like (ra, rb, rc) .

Practically, this step involves the construction a chain of register numbers from the register-pair frequency matrix by greedily selecting the most common pairs first and adding them to the chain under certain rules. Selection of the more frequent pairs means that the more instructions that exist which are isomorphic, but differ by the register pair $[ra, rb]$, the more important it is that ra and rb be encoded as register numbers with a small Hamming distance between them. As each new register pair is selected from the matrix and reviewed for addition to the chain, three possible scenarios arise.

Step 3.1: If the pair ra, rb are not found in the chain, add a new, disjointed chain

Step 3.2: If one of ra and rb is found, and is located at the end of a chain, add the other register to the end of that chain. If the found register is not located at the end of a chain, do nothing.

Step 3.3: If both ra and rb are found, and they are both located at the end of two separate chains, join those chains together. If both are found, but not located at the end of two separate chains, do nothing.

Step 4: Assign register numbers

Assign registers a new register number based on the newly constructed register chain and the Gray coding sequence. This ensures that every pair of registers (ra, rb) that are adjacent in the register chain will have a Hamming distance: $HD(ra, rb) = 1$. Any registers that, during this process, were not added to the chain, are assigned the remaining register numbers in any order (though practically, this is unusual). The process of assigning Gray code values to registers in the chain is depicted also in Figure 3, with two examples (Ex1 and Ex2) of Gray code sequences. In both examples, each adjacent register pair in the chain will differ by only one bit.

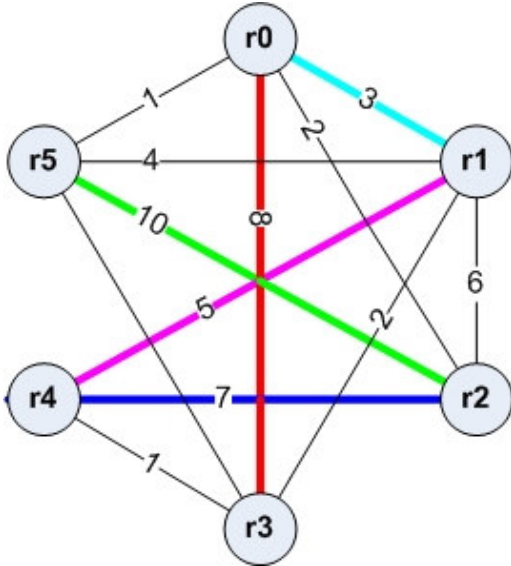
Step 5: Applying new Register Numbers

Process the instructions again to reassign the registers and output to a file.

3.3 Benchmarks Used and Compiler Optimizations

Once the original (compiler output) file has had its registers reassigned using firstly the Gray Code Reassignment method and secondly the Register-Pair Frequency Reassignment method, all three binaries were compressed using the Hamming distance based compression scheme and results noted.

The benchmarks tested were taken from both the MediaBench [1] and Spec2000 [2] suites. All benchmarks were compiled for the Texas Instruments TMS320C6700 [21] with various optimizations and the smallest code size chosen, consistent with our previous work. The benchmarks processed include: `cjpeg`,



$(r2, r5) \Rightarrow (r2, r5)$
 $(r0, r3) \Rightarrow (r2, r5) (r0, r3)$
 $(r2, r4) \Rightarrow (r4, r2, r5) (r0, r3)$
 $(r1, r2) \Rightarrow (r4, r2, r5) (r0, r3)$
 $(r1, r4) \Rightarrow (r1, r4, r2, r5) (r0, r3)$
 $(r1, r5) \Rightarrow (r1, r4, r2, r5) (r0, r3)$
 $(r0, r1) \Rightarrow (r5, r2, r4, r1, r0, r3)$

	Ex1	Ex2
r5 (00101)	- 00000	- 00101
r2 (00010)	- 00001	- 00100
r4 (00100)	- 00011	- 00000
r1 (00001)	- 00010	- 00001
r0 (00000)	- 00110	- 00011
r3 (00011)	- 00111	- 00010

Figure 3 – A Graph Representation Example of Register Chain Construction

djpeg, epic, mpeg2dec, mpeg2enc and unepic from MediaBench and ammp, art, equake, mcf, parser and twolf from Spec2000.

3.4 Compression Scheme Used

The compression scheme used is based on vector Hamming distances between instructions (vectors). This approach to code compression originated in [18] but was investigated further in [20]. It involves the appropriate selection of dictionary vectors such that all program vectors are at most a specified Hamming distance from a dictionary vector (the Hamming distance between two vectors is the number of bits that are different). This method means that two vectors that differ by only one bit will not require both vectors to be stored in the dictionary. One of the two vectors is stored and the other merely references the stored vector and points out which bit needs to be toggled. Extra bit toggling information is required after the dictionary references to accurately restore original code.

In this compression scheme, the binary to be compressed is read in a vector at a time and a frequency distribution histogram of the used vector space is constructed. The list of unique instruction vectors is called the dictionary. From this dictionary, a subset of vectors (called the reduced dictionary) is selected such that all original dictionary vectors are at most a set Hamming distance from any one of the reduced dictionary vectors. The binary is then re-encoded using the reduced dictionary entries with extra information for any bits that require toggling. A Hamming distance upper limit of 3 is used, as that was found to produce the best compression ratios in [20]. The required hardware decompression unit is shown in Figure 4.

4. RESULTS

Although compression ratio is the main metric to be minimized in this work, other metrics are also reported for completeness. One measure of the effect the register reassignment has had on the Hamming distance based compression, is to look at the number of

reduced dictionary entries (vectors) required such that every instruction in the program is within the specified Hamming distance of at least one of the reduced dictionary vectors. Table 1 shows each benchmark together with the number of reduced vectors needed for the compression of the original binary, the binary after Gray code reassignment and the binary after register-pair reassignment.

The *Reduction* column in Table 1 demonstrates the change in the number of reduced dictionary vectors from the original binary to the number in the register-pair frequency binary. These benchmarks produced an average reduction of dictionary vectors of 9.74%, with some benchmarks up to 15% reduction.

Information regarding the specific counts of program vectors at a Hamming distance of 0, 1, 2 and 3 from the reduced dictionary vectors for all three binary types is also a measure of the effect of the reassignment schemes. An example of the number of program vectors found certain Hamming distances from an entry in the dictionary is shown in Table 2 using the example benchmark ammp.

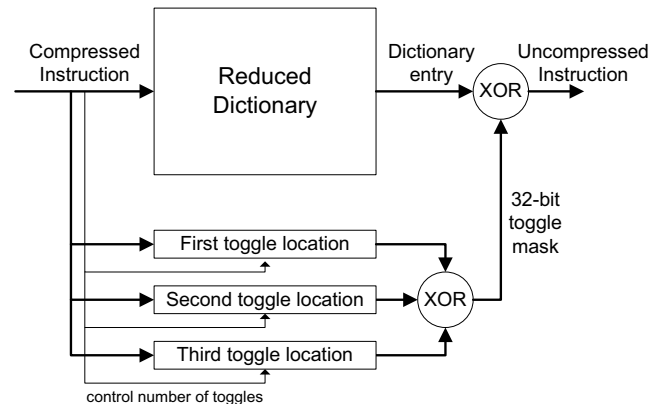


Figure 4 – Decompression Hardware Block Diagram

Table 1 - Number of Unique/Reduced Vectors in Compression of Original, Gray Code and Register-Pair Frequency Binaries.

	Size (bytes)	Unique Vectors	Reduced Vectors			Reduction
			Original	Gray Coded	Register-Pair	
ammp	172112	13771	1817	1773	1708	6.00%
art	58144	6657	1424	1272	1348	5.34%
cjpeg	129792	12865	2022	1700	1725	14.69%
djpeg	115424	11805	2077	1781	1756	15.45%
epic	31040	3455	688	609	600	12.79%
equake	67680	7935	1619	1482	1512	6.61%
mcf	56960	6883	1540	1380	1399	9.16%
mpeg2dec	48928	5866	1158	1028	1070	7.60%
mpeg2enc	75072	8200	1398	1196	1237	11.52%
parser	161856	13769	2267	2067	2151	5.12%
twolf	281824	24192	3307	2944	2968	10.25%
unepic	18560	2412	546	498	496	9.16%

This example benchmark has 43028 program instructions (vectors), of which 8773 are located in the dictionary (HD0 = “Hamming Distance of 0”), 11677 are a Hamming distance of 1 away from at least one entry in the dictionary (HD1), 16520 are a Hamming distance of 2 (HD2) and 6058 are a Hamming distance of 3. These statistics correspond to the original binary output by the compiler.

When the register-pair frequency reassignment algorithm is applied, the number of program vectors now located in the reduced dictionary (i.e. within a Hamming distance of 0) has increased by 58%. This result means that the reduced dictionary selection method favors the arrangement of registers within instructions output by the register-pair reassignment algorithm.

4.1 Gray Code Reassignment Results

Initially, a static register sequence reassignment was tested to determine if reassignment of register indices could influence the overall compression ratio obtained by the Hamming distance based compression scheme. The Gray code sequence was a good choice to implement.

Figure 5 shows the sizes of the compressed binaries for each benchmark. The size of the binary after Gray code register reassignment is the middle column in each benchmark group. In all cases, the size was smaller, but only slightly, averaging a reduction in compression ratio of ~1%. Some benchmarks did

give more reduction, up to 2% and 5%. Figure 6 shows the compression ratio reduction for each benchmark.

4.2 Register-Pair Frequency Reassignment Results

To investigate whether a program-aware register reassigning algorithm would improve the code compression, the Register-Pair Frequency Reassignment technique described in Section 3.2 was implemented.

Figure 5 shows the sizes of the compressed binaries for each benchmark. The size of the binary after register-pair frequency reassignment is the third column in each benchmark group. It is clear that in all cases, the compressed binary size for the register-pair frequency reassignment is smaller than the other two compressed binaries. An average compression ratio *reduction* of 3-4% was experienced after the register-pair frequency reassignment, with one benchmark, `djpeg`, exhibiting a compression ratio reduction of nearly 7%. The compression ratio obtained by compressing the original `djpeg` binary was 73.48%. Using the register-pair frequency to reassign register, the compression ratio dropped to 66.51%.

5. CONCLUSIONS AND FUTURE WORK

The experiments in this paper showed that techniques such as

Table 2 – Hamming Distance Frequency for ammp Example

	Original	Gray Coded	Register-Pair
0HD	8773	9177 (+4.61%)	13906 (+58.51%)
1HD	11677	11739 (+0.53%)	8351 (-28.48%)
2HD	16520	15805 (-4.33%)	14746 (-10.74%)
3HD	6058	6307 (+4.11%)	6025 (-0.54%)
	43028	43028	43028

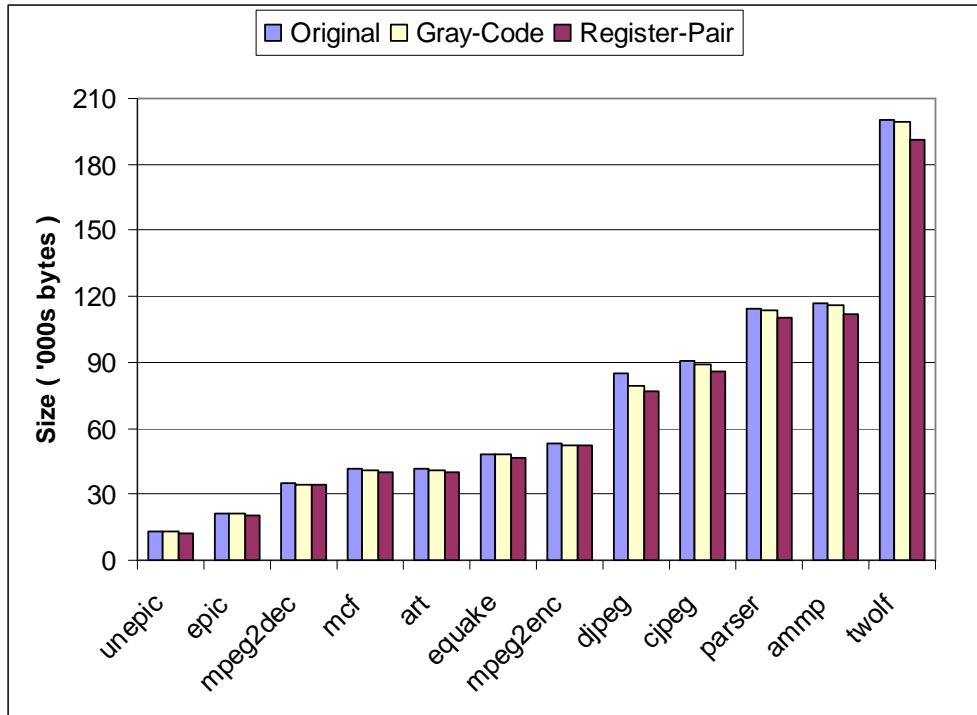


Figure 5 - Sizes of Compressed Binaries

register reassigning can be used to prepare compiler binaries for subsequent compression. We showed that general purpose register reassignment is a technique that allows for gains in compression efficiency with no penalty in hardware.

As the compression scheme used in the paper selects instructions for a dictionary based on the vector Hamming distances between instructions, it is no surprise that a binary translation that

minimizes Hamming distances would result in more compression.

Although the static Gray code sequence register reassignment technique did affect the overall size of the compressed binary, this difference was only very slight. One benchmark did experience significant reduction in compression ratio, ~5%, however, it is unlikely that a static sequence could be generated that affected a majority of benchmarks in the same way.

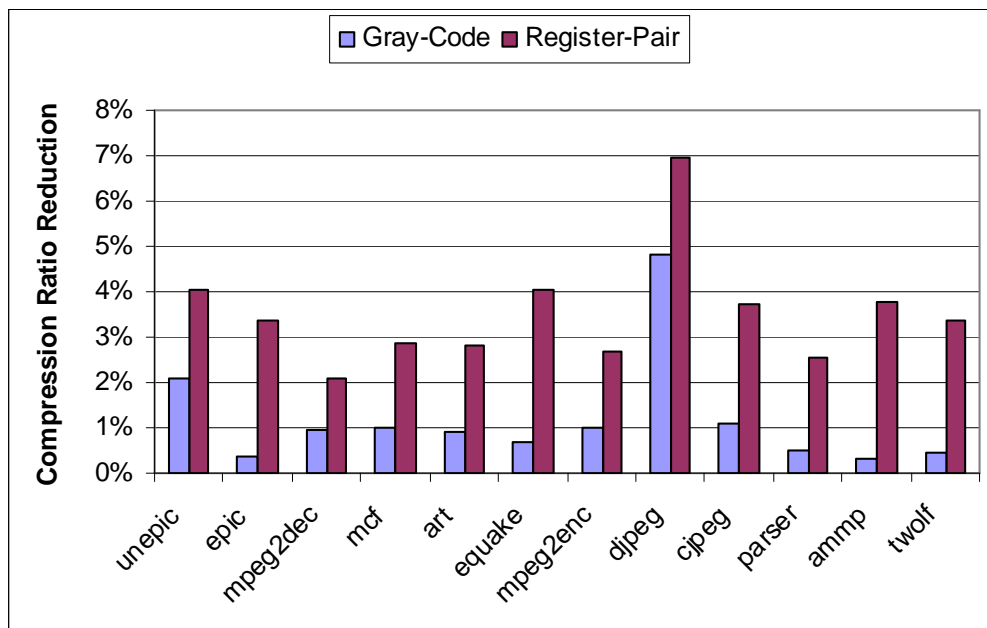


Figure 6 – Compression Ratio Reduction due to Gray Code and Register-Pair Reassignment

A program-specific register reassignment is certain to produce the best results. The algorithm described in this paper greedily adds frequent register pairs to a chain that is later encoded using the Gray code sequence. This technique reduced the number of dictionary vectors required by the compression scheme by an average of 9.74%. Also, code compression ratios of register-reassigned binaries were consistently around 3-4% lower than code compression applied to original binaries, with the highest such reduction at nearly 7%.

Further work includes the investigation into other binary translation techniques that could be used to increase code compression efficiency. Some suggested techniques include commutative register switching (switching the order of input registers for commutative instructions), dead register detection and assignment (some work has been done in this area in the related research into power savings), and finally, a complete register re-allocation and selection of program values for spilling and splitting.

6. REFERENCES

- [1] "Mediabench Benchmarks", <http://cares.icsl.ucla.edu/MediaBench/>
- [2] "SPEC CPU2000 Benchmarks", <http://www.spec.org/cpu2000/>
- [3] Atmel-Corporation, "AT572D740 Summary (Datasheet)", http://www.atmel.com/dyn/resources/prod_documents/7001s.pdf
- [4] P. Centoducatte, G. Araujo, and R. Pannain, "Compressed code execution on DSP architectures," in Proceedings 12th International Symposium on System Synthesis. 1999: IEEE Comput. Soc, Los Alamitos, CA, USA, 1999, pp. 56-61.
- [5] K. D. Cooper and N. McIntosh, "Enhanced code compression for embedded RISC processors," in SIGPLAN Notices. May 1999; 34(5): ACM, 1999, pp. 139-49.
- [6] J. Ernst, W. Evans, et al., "Code compression," in SIGPLAN Notices. May 1997; 32(5): ACM, 1997, pp. 358-65.
- [7] IBM, CodePack(TM) PowerPC Code Compression Utility User's Manual Version 4.1: IBM, 1998.
- [8] N. Ishiura and M. Yamaguchi, "Instruction Code Compression for Application Specific VLIW Processors BAsed on Automatic Field Partitioning," 1997.
- [9] S. Y. Larin and T. M. Conte, "Compiler-driven cached code compression schemes for embedded ILP processors," in MICRO 32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture. 1999: IEEE Comput. Soc, Los Alamitos, CA, USA, 1999, pp. 82-92.
- [10] C. Lefurgy, P. Bird, I. C. Chen, and T. Mudge, "Improving code density using compression techniques," in Proceedings. Thirtieth Annual IEEE/ACM International Symposium on Microarchitecture Cat. No.97TB100184. 1997: IEEE Comput. Soc, Los Alamitos, CA, USA, 1997, pp. 194-203.
- [11] C. Lefurgy and T. Mudge, "Code Compression for DSP", Compiler and Architecture Support for Embedded Computing Systems, George Washington University, Washington DC, 1998.
- [12] C. Lefurgy, E. Piccininni, and T. Mudge, "Reducing code size with run-time decompression," in Proceedings Sixth International Symposium on High Performance Computer Architecture. HPCA 6 Cat. No.PR00550. 1999: IEEE Comput. Soc, Los Alamitos, CA, USA, 1999, pp. 218-28.
- [13] H. A. Lekatsas, "Code compression for embedded systems," Princeton University, 2000, pp. 171.
- [14] S. Liao, S. Devadas, and K. Keutzer, "A text-compression-based method for code size minimization in embedded systems," ACM Transactions on Design Automation of Electronic Systems, vol. 4, pp. 12-38, 1999.
- [15] S. S. Muchnick, Advanced compiler design and implementation: Morgan Kaufmann Publishers Inc., 1997.
- [16] S. J. Nam, In Cheol Park, and Chong Min Kyung, "Improving dictionary-based code compression in VLIW architectures," IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences, vol. E82-A, pp. 2318-24, 1999.
- [17] P. Petrov and A. Orailoglu, "Transforming binary code for low-power embedded processors," Micro, IEEE, vol. 24, pp. 21-33, 2004.
- [18] J. Prakash, C. Sandeep, P. Shankar, and Y. N. Srikant, "A Simple and Fast Scheme for Code Compression for VLIW processors," in Proceedings DCC 2003. Data Compression Conference, J. A. Storer and M. Cohn, Eds.: IEEE Comput. Soc, Los Alamitos, CA, USA, 2003, pp. 444.
- [19] M. Ros and P. Sutton, "Compiler optimization and ordering effects on VLIW code compression," in Proceedings of the international conference on Compilers, Architectures and Synthesis for embedded systems. San Jose: ACM Press, 2003, pp. 95--103.
- [20] M. Ros and P. Sutton, "A hamming distance based VLIW/EPIC code compression technique," in Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems. Washington DC, USA: ACM Press, 2004, pp. 132-139.
- [21] Texas-Instruments, "TMS320 DSP Algorithm Standard Rules and Guidelines", <http://focus.ti.com/lit/ug/spru352e/spru352e.pdf>
- [22] A. Wolfe and A. Chanin, "Executing compressed programs on an embedded RISC architecture," in SIGMICRO Newsletter. Dec. 1992; 23(1 2), 1992, pp. 81-91.
- [23] Y. Xie, H. Lekatsas, and W. Wolf, "Code compression for VLIW processors," in Proceedings DCC 2001. Data Compression Conference. 2001, J. A. Storer and M. Cohn, Eds.: IEEE Comput. Soc, Los Alamitos, CA, USA, 2001, pp. 525.
- [24] Y. Xie, W. Wolf, and H. Lekatsas, "Code compression for VLIW processors using variable-to-fixed coding," in 15th International Symposium on System Synthesis IEEE Cat. No.02EX631. 2002: ACM, New York, NY, USA, 2002, pp. 138-43.
- [25] K. Zhang, T. Zhang, and S. Pande, "Binary translation to improve energy efficiency through post-pass register re-allocation," in Proceedings of the fourth ACM international conference on Embedded software. Pisa, Italy: ACM Press, 2004, pp. 74-85.