Instruction Merging and Multimedia Technology

Matt McCormick
mattmcc@cs.wisc.edu

Computer Sciences Department
University of Wisconsin
1210 West Dayton Street
Madison, Wisconsin 53706, U.S.A.

1. Introduction

In recent years several microprocessor manufacturers have begun to release new hardware products that allow multiple pieces of data to be packed into a single register and then with one instruction do work on all of that data. The concept is simple, if most of the data is 8-bits long (or 16 or 32) why not pack 8 (or 4 or 2) pieces of data into a 64-bit register and achieve considerably higher performance from a given number of instructions. This is often referred to as single instruction, multiple data (SIMD) [Flynn]. The main purpose of these new architectures is to enhance multimedia applications. While these architectures may provide the greatest opportunity for improvement in this environment [PCmag], there is no reason to limit the application of such architecture to merely multimedia applications. Consider the fact that many programs are written assuming the use of 32-bit integers. This means that many math and logical operations in a program can potentially be combined with a second instruction doing the same type of operation. If the programmer intended to use short (16-bit) or byte size integers, the potential number of instructions that can be combined should be even greater – and not just for multimedia applications. This paper presents an algorithm for a back-end compiler to search through a set of instructions and determine possible instruction merging opportunities.

The following section introduces the algorithms used to find all of the mergeable instructions, for determining which instructions *can* be merged together and which instructions *will* be merged together. Section 3 presents and discusses the results of this research, and Section 4 gives some concluding remarks.

2. Algorithm

2.1 Basic Data Structures

The two most difficult parts in merging instructions is determining which instructions *can* be merged and, if multiple options are present, which instructions *to* merge. Figure 1 shows the data structures and a flow chart of the operations necessary to perform this analysis.

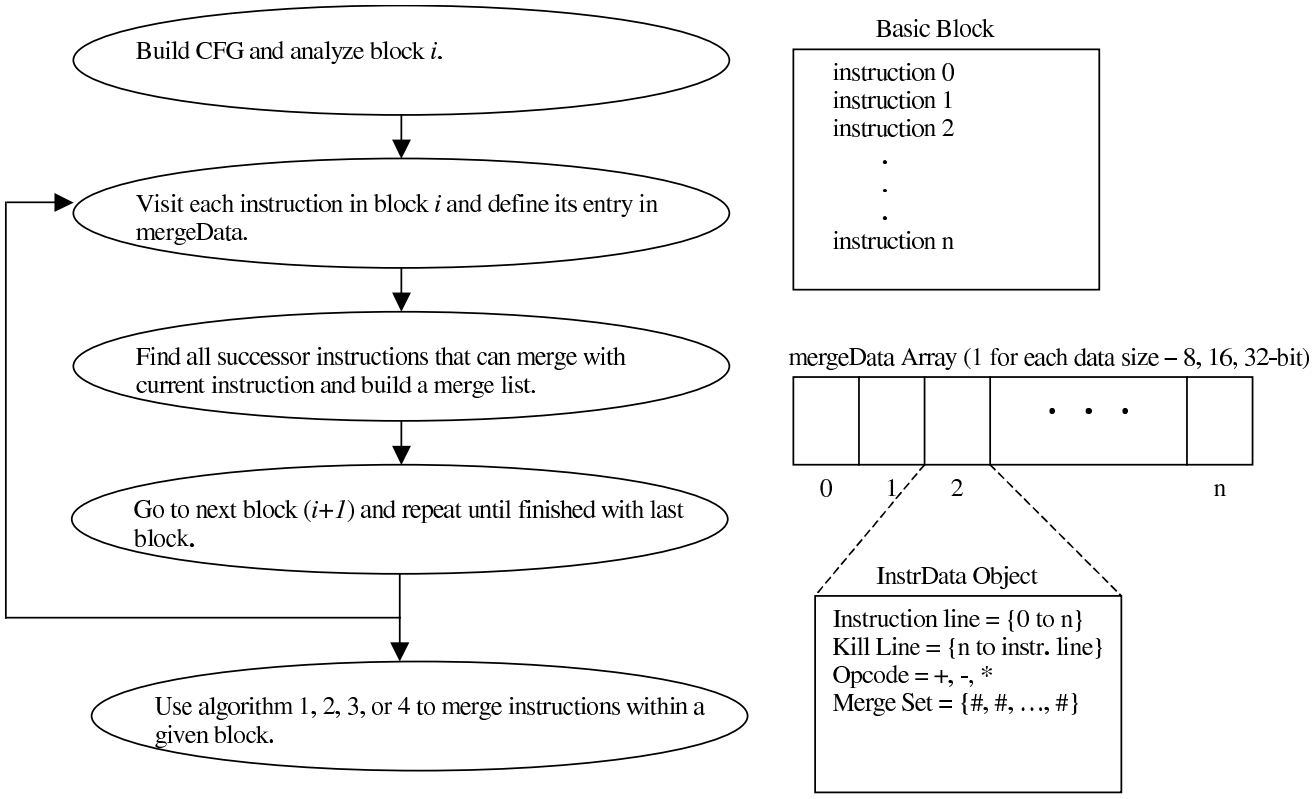

Figure 1

The left hand side is a flow chart of the operations necessary to build the data structures on the right hand side of the diagram. All analysis is done on a per basic block basis.

The most important data structure to consider is the *InstrData* object. It contains the following information:

1. Instruction Line – the line number in the basic block that the mergeable instruction is found on. The range is from 0 to n (the total number of instructions in the block).
2. Kill Line – the first line in the block *following* the current instruction line that this instruction cannot be moved passed due to data dependencies. The range will either be the line immediately after the current line to the end of the block or –1. –1 is used if this instruction can be moved to any of the following lines in the block.
3. Opcode – the opcode of this mergeable instruction. For this research the only operations considered for merger were +, -, or *.
4. Merge Set – an array of all the mergeable instructions *prior* to the current instruction that can be merged with the current instruction. Note that all the entries in the merge set must have a kill line value greater than the current line.

Before proceeding, it would be useful to define a few terms used throughout the rest of this paper. Mergeable instructions are all instructions considered for merging. For this research these include ADD, SUB, and MULT of signed and unsigned integers only. The Intel architecture also allows DIV, some logical operators, and some shift operators to be used with multiple data but those instructions are not considered here. The only other term to be defined is the kill line. This is a line in the basic block to which an instruction under consideration cannot be moved passed due to data dependencies. If no kill line for an instruction exists within the block, this value will be set to –1.

2.2 Defining the Merge Set

The methodology for determining which instructions to merge is actually quite simple. The first step is to build a control flow graph (CFG) where each node represents a basic block. Secondly, an array of *InstrData* objects (called *mergeData*) is created for each data size being considered. Data sizes of 8, 16, and 32 bits, signed and unsigned, were considered for a total of six arrays. To initialize each *InstrData* object in *mergeData*, each instruction in the basic block is visited. If it happens to be a mergeable instruction, its line number, opcode, and kill line are determined and recorded. No information about which instructions an entry can merge with is determined on this first pass. In other words, after each entry in *mergeData* has been initialized, there will be no data recorded in the merge set of any *InstrData* object. This will be determined in the second pass through *mergeData*.

The purpose of the second pass through the data is to examine the instruction represented in *mergeData[i]* and find all instructions *j* occurring before *i* that *i* can be merged with. As an example, assume that every instruction in a block can be merged with every other instruction in the block (the kill line value for each entry in *mergeData* is –1 after the first pass). This would mean that the size of any entry's merge set would be 1 bigger than the merge set size of the entry preceding it in the array. In fact, if every entry could be merged with every other entry, the merge set size of any given entry in *mergeData* would be exactly equal to the entry's index number in the *mergeData* array.

So how exactly is the merge set for a particular entry determined? If element *i* is being analyzed, then every element *j* preceding it in the *mergeData* array that has the same opcode and whose kill line is greater than *i*'s line number is a candidate for merging with *i*. Hence, *j* is added to the end of *i*'s merge set.

2.3 Merging the Instructions

It is now possible begin merging instructions, but two important questions must be considered first. If a particular entry in *mergeData* can be merged with more instructions than its data size

allows, which ones should it merge with? If not every instruction can be merged, which ones should be? Deciding the optimal solution to these problems appears to be NP complete. For example, if instruction *i* is a 32-bit instruction and it has 5 possible 32-bit instructions it can be merged with and each of those has 5 instructions they can merge with, and so on; the possible solutions to the best merges grows exponentially. It is interesting to note, however, that given the way the merge set for each instruction is set-up, it may be possible to search the entire space (although this was not tried in this research). The reason for this is because the merge set for an instruction only considers those instructions prior to it. Hence, the size of the merge set for instructions closer to the start of the block should tend to be less than those mergeable instructions near the end of a block. In fact, the first mergeable instruction will always have a merge set size of zero – even if it can merge with every instruction following it! Thus the size of the merge set may be manageable for an exhaustive search.

The research presented here considered four different heuristics for determining the "best" instruction merge combinations. Each is broken down in to two parts – the location in the block of the instruction to be merged and the location of the instruction to merge with. The breakdown is as follows:

1. **earliest – earliest:** start at mergeData[0], merge with first element(s) in merge set
2. **earliest – latest:** start at mergeData[0], merge with the last element(s) in merge set
3. **latest – earliest:** start at mergeData[n], merge with the first element(s) in merge set
4. **latest – latest:** start at mergeData[n], merge with the last element(s) in merge set

After *mergeData* and the merge set for each mergeable instruction has been set-up, any of the heuristics from 1 to 4 is straightforward to implement. Each simply involves two loops (one nested in the other) to search through the two arrays mentioned. Because code is being moved to a new location, it is important to make sure that all data dependencies are taken care of. However, remember the merge set for any instruction *i* was constructed in such a way that it guaranteed that any entry *j* in its merge set could be moved to *i* without corrupting or being corrupted by any instruction between *i* and *j*. So deciding which instructions can be moved is a trivial matter after the merge sets have been constructed. Psuedo-code for method 3 above is listed in Figure 2 below. Code for the other 3 heuristics would be very similar.

```
// preliminary work
set-up merge data
for(each entry in merge data)
        define the merge set
// merge instructions
for(i=size(mergeData)-1; i>=0; i--) {
                mergeInstruction = mergeData[i]
        // if data size is 8, 16, or 32; need to find another 7, 3, or 1, respectively
        for(j=0; j<numToMerge(data size)-1 && notEmpty(mergeSet); j++) {
                remove mergeSet[j]
                 add mergeSet[j] to merging
        }
        // make sure each instruction to be merged is removed from mergeData
        // so that it won't be considered again for merging and then merge
        removeFromMergeData(merging)
        mergeToLocationOfMergeInstruction(merging)
}
```

Figure 2
Psuedo-code for implementing latest – earliest merging heuristic

These 4 heuristics were chosen several reasons. First of all, they should be very fast in software because no decisions have to be made – simply grab as many elements from the merge set as are needed by an instruction. While this project did not gather the data to prove which of these algorithms would perform best, it would seem that the either 2 or 4 should outperform the rest and perform comparable to one another. The reason is because they give the instructions closest to the front of the block as much freedom as possible in making selections. Because of the way the algorithm adds instructions to the merge set, the instructions at the end of the merge set are farthest away from the front of the block. Earliest – earliest allows instructions at the front of *mergeData* to grab from the front of the merge set and assures that early instructions will have as much selection as possible. When using latest – latest, the instructions near the front of the block should still have good selections available because instructions in the rear of *mergeData* are trying to merge with instructions at the rear as well. Following this reasoning, it would be expected that earliest – latest would perform slightly worse than latest – earliest, and latest – earliest would be expected to perform the worst.

3. Results

The research presented in this paper was conducted using an Intel Pentium III™ processor with the MMX technology running programs from the Spec95 integer benchmarks. Table 1 shows a count as to the total number of instructions in each of these benchmarks that would be available for merging with another instruction. Only ADD, SUB, and MULT instructions are included in this data, but it shows that significant opportunities are available for merging instructions. The

data shows that up to 4.22% of all the instructions in a program could potentially be combined with *at least* one other instruction; however, it can also be seen from the number actually merged that data and control dependencies severely limit merging possibilities. Since the actually merging algorithm was not implemented, the number of instructions merged is not based on actual mergers. Rather, it the total number of instructions that were added to the merge set of some other instruction.

TABLE 1
Relationship between 5 integer benchmarks and how many instructions are mergeable and how many are actually merged.

Benchmark	Total Instructions	# Mergeable	% Mergeable	# Merged	% Merged
compress	3073	138	4.49%	5	0.16%
vortex	170060	3384	1.99%	987	0.58%
m88ksim	83259	2574	3.09%	224	0.27%
jpeg	102834	6720	6.53%	1677	1.63%
go	175256	9749	5.56%	748	0.43%
TOTAL	**534482**	**22565**	**4.22%**	**3641**	**0.68%**

Table 2 is a breakdown of the data sizes of each of these mergeable instructions. The data size of an instruction is chosen to be that of its largest operand. From Table 2 it can be seen that almost all of the mergeable instructions involved a standard, signed *int*. This is interesting because [Mestan] shows that on average close to 50% of the instructions that pass through the integer functional units in a pipelined, superscalar processor are 16 bits or less. This means that the numbers for the 16-bit merging data would increase significantly if some type of maximum value analysis were done to determine at each use of a variable how many bits would be needed to represent it. The algorithms presented in section 2 above merely consider the data type as declared by a programmer. The reason it would be beneficial to know how many bits are needed is because the fewer the bits, the more data that can be packed into an instruction, and the more work that can be done in a single instruction.

benchmark	32-bit unsigned	32-bit signed	16-bit unsigned	16-bit signed	8-bit unsigned	8-bit signed
compress	3	128	0	0	5	2
vortex	2785	794	31	6	8	3
m88ksim	1127	1413	4	0	9	21
jpeg	2654	3901	8	5	141	11
go	875	8860	0	0	12	2
TOTAL	**7444**	**15096**	**43**	**11**	**175**	**39**
%	32.64%	66.19%	0.19%	0.05%	0.77%	0.17%

TABLE 2
Breakdown of all mergeable instructions by type

The reason such a small percentage of the overall mergeable instructions are merged is because the vast majority of them are killed shortly after they execute. This makes sense since the algorithm used for merging is only considering a single basic block and because most instructions are calculated as they are needed. If the algorithm were modified to consider all equivalent blocks, many more instructions would be available for merging. Few modifications would be needed to implement this change and it would most likely result in many more instructions available for co-scheduling.

The data presented here only considers six types of instructions for merging (ADD, SUB, MULT, signed and unsigned for each) and applies no anti-aliasing techniques for working with pointers. This means the results presented here lean heavily toward the pessimistic side. A future implementation of this work would most surely involve more instructions, cross block code motion, anti-aliasing, and maximum (and minimum) value prediction. Future work would also involve the actual implementation and evaluation of changing the source code to use SIMD instructions.

4. Conclusions

The material presented here shows that SIMD architectural designs intended for multimedia applications could have substantial impact on the performance of non-multimedia intensive programs. Over 4% of all the instructions examined in the Spec95 integer benchmark sweet are possibilities for merging. This of course is highly dependant on the type of program, but with proper analysis technique nearly every program can find some benefit from merging instructions. The key is to identify which instructions can be merged, how many instructions can be merged

into a single instruction, and where that instruction will be placed to optimize code motion without compromising data and structural hazards.

5. Bibliography

[Flynn] Flynn, M. J., *Very High-Speed Computing Systems*, Proceedings of the IEEE 28(4)23-31, 1995.

[PCmag] Metz, C., *MMX Extends the Desktop*, PC Magazine, Volume 16, No. 3, Feb. 4, 1997.

[Mestan] Mestan, B and Hart, S., *A Value Sensitive Architecture*, Dec., 2000.