Bitwidth Aware Global Register Allocation

Sriraman Tallam Rajiv Gupta
Department of Computer Science
The University of Arizona
Tucson, Arizona 85721
{tmsriram,gupta}@cs.arizona.edu

ABSTRACT

Multimedia and network processing applications make extensive use of subword data. Since registers are capable of holding a full data word, when a subword variable is assigned a register, only part of the register is used. New embedded processors have started supporting instruction sets that allow direct referencing of bit sections within registers and therefore multiple subword variables can be made to simultaneously reside in the same register without hindering accesses to these variables. However, a new register allocation algorithm is needed that is aware of the bitwidths of program variables and is capable of packing multiple subword variables into a single register. This paper presents one such algorithm.

The algorithm we propose has two key steps. First, a combination of forward and backward data flow analyses are developed to determine the bitwidths of program variables throughout the program. This analysis is required because the declared bitwidths of variables are often larger than their true bitwidths and moreover the minimal bitwidths of a program variable can vary from one program point to another. Second, a novel interference graph representation is designed to enable support for a fast and highly accurate algorithm for packing of subword variables into a single register. Packing is carried out by a node coalescing phase that precedes the conventional graph coloring phase of register allocation. In contrast to traditional node coalescing, packing coalesces a set of interfering nodes. Our experiments show that our bitwidth aware register allocation algorithm reduces the register requirements by 10% to 50% over a traditional register allocation algorithm that assigns separate registers to simultaneously live subword variables.

Categories and Subject Descriptors

C.1 [**Computer Systems Organization**]: Processor Architectures; D.3.4 [**Programming Languages**]: Processors—*compilers*

General Terms

Algorithms, Measurement, Performance

Keywords

subword data, minimal bitwidth, packing interfering nodes, embedded applications

POPL'03, January 15–17, 2003, New Orleans, Louisiana, USA.

1. INTRODUCTION

Programs that manipulate data at subword level, i.e. bit sections within a word, are common place in the embedded domain. Examples of such applications include media processing as well as network processing codes [12, 19]. A key characteristic of such applications is that at some point the data exists in packed form, that is, multiple data items are packed together into a single word of memory. In fact in most cases the input or the output of an application consists of packed data. If the input consists of packed data, the application typically unpacks it for further processing. If the output is required to be in packed form, the application computes the results and explicitly packs it before generating the output. Since C is the language of choice for embedded applications, the packing and unpacking operations are visible in form of bitwise logical operations and shift operations in the code. In addition to the generation of extra instructions for packing and unpacking data, additional registers are required to hold values in both packed and unpacked form therefore causing an increase in *register pressure*.

New instruction set architectures for embedded and network processors allow bit sections within a register to be directly referenced [6, 15, 18]. For example, the following instruction adds a 4 bit value from $R2$ with a 6 bit value from $R3$ and stores a 8 bit result in $R1$. The operands are extended by adding leading zero bits to match the size of the result before the addition is carried out.

$$R1_{0..7} \leftarrow R2_{0..3} + R3_{0..5}$$

In our recent work we incorporated bit section referencing into the popular ARM processor. We have shown that the proper use of such instructions eliminates the need for explicit packing and unpacking operations and thus reduces the number of executed instructions significantly [13]. Another important consequence of having this instruction set support is that multiple subword sized variables can be made to simultaneously reside in the same register without hindering access to the variables. Thus this approach reduces register requirements of the program. Since embedded processors support a small number of registers (e.g., ARM [16] supports 16 registers and even fewer are directly accessible by most instructions in Thumb mode) efficient use of register resources is extremely important.

To illustrate the potential for reduction in register requirements, let us consider the examples shown in Fig. 1 that are typical of embedded codes. These code fragments, taken from the `adpcm` (audio) and `gsm` (speech) applications respectively, perform unpacking and packing. Each code fragment references three variables which have the declared size of 8 bits each. The live ranges of the variables, including their widths, are shown. By examining these live ranges we find that a traditional register allocator must use two registers to hold their values. However, bitwidth aware register allocation can dramatically reduce the register requirements. We can

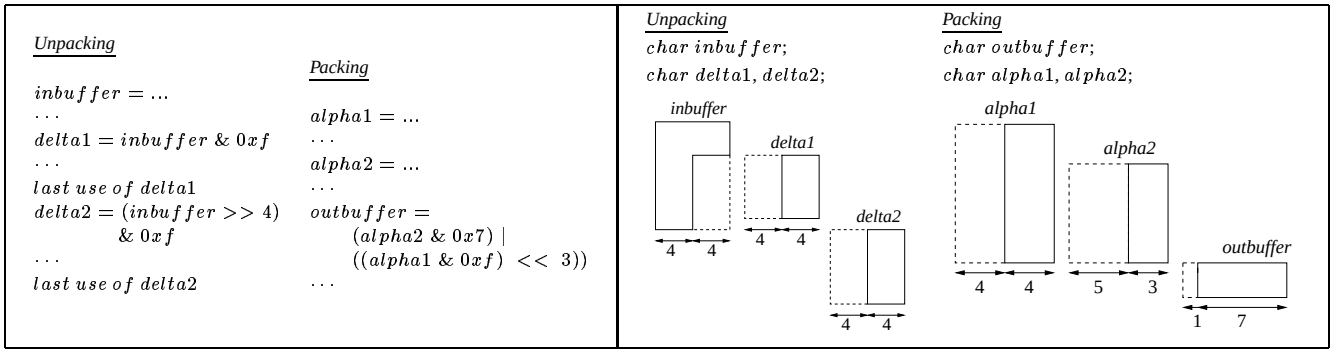


Figure 1: Subword variables in multimedia codes.

hold the values of these variables in 8 bits and 7 bits of a single register respectively for the two code fragments. The remaining bits of this register can be used to hold additional subword sized variables.

In this paper we describe an approach for achieving register allocations that use a part of a single register, as opposed to multiple registers, for the above code fragments. There are two key components of our approach. First, we employ algorithms for constructing live ranges of variables such that the minimal bitwidths, or widths for short, of the live ranges at various program points are also computed. Second, we employ a fast and effective method for packing together multiple live ranges. The packing phase essentially performs coalescing of interfering nodes on an enhanced interference graph representation for the program. Following packing, register allocation is carried out using a conventional graph coloring algorithm that assigns a single register to each node in the graph [1, 3, 8]. There are a number of challenges of developing fast and yet effective algorithms for each of the above components that are described below:

Live range construction

The first challenge is to identify the *minimal width* of each live range at each relevant program point. Analysis must be developed for this purpose due to two reasons that are illustrated by our examples: variables are declared to be of larger than needed bitwidths (e.g., `delta1` is declared as an 8 bit entity while it only uses 4 bits); and the bitwidth of a variable can change from one program point to another as a variable may contain multiple data items which are consumed one by one (e.g., `inbuffer` initially is 8 bits of data, after `delta1` is assigned it contains only 4 bits of useful data). We present a combination of forward and backward data flow analysis to find the minimal widths.

The second challenge is to *efficiently* identify the minimal widths. An obvious way is to develop an analysis which, for a given variable that is declared to be b bits wide, determines the "need" for keeping each of the b bits in a register at each program point. The cost of such bitwise analysis will be high as it is directly dependent upon the bitwidths of the variables. To achieve efficiency, we develop an analysis which views each variable, regardless of its size, as made up of three bit sections called the leading, middle, and trailing sections. The goal of the analysis is to determine the minimal sized middle section that must be kept in the register while the leading and trailing sections can be discarded. This approach is effective in practice because the unneeded bits of a variable at a program point typically form leading and/or trailing bit sections.

Packing multiple variables into a register

When variables are packed together through coalescing of two nodes in the interference graph, the shapes of the live ranges must be taken into account to determine whether or not the live ranges can be coalesced, and if coalescing is possible, the characteristics of the coalesced live range must be determined to perform further coalescing. A simple approach to this problem may not be *accurate* leading to missed opportunities for coalescing. For example, in our earlier example the maximum width of `inbuffer` and `delta1` were 8 and 4 bits respectively. A simple method that ignores their shapes and assigns a width of 12 bits to the live range resulting from coalescing the two, overestimates the width by 4 bits. Therefore while this approach is simple, and thus fast, it will miss coalescing opportunities.

A completely accurate approach can be developed which compares the shapes of the live ranges at all relevant program points to determine whether they can be coalesced, and if that is possible, it computes the compact shape of the resulting live range. While this approach will not miss coalescing opportunities, it is too expensive. We present a *fast* and *highly accurate* approach for node coalescing based upon an enhanced *labeled interference graph*. The shapes of interfering live range pairs are compared exactly once to generate the labelling. Node coalescing is driven by the labelling which is updated in constant time following each coalescing step. While the labelling is approximate, it is highly accurate in practice and therefore missed coalescing opportunities are rare.

Outline

The remainder of the paper is organized as follows. In section 2 we present the live range construction algorithm. The enhanced interference graph representation and the node coalescing algorithm based upon it to affect variable packing is described in section 3. Experimental evaluation is presented in section 4. Related work is discussed in section 5 and conclusions are given in section 6.

2. LIVE RANGE CONSTRUCTION

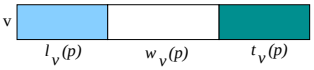
Definition 1. (Live Range) The *live range* of a variable v is the program region over which the value of v is live, that is, for each point in the live range, a subset of bits in v's current value may be used in a future computation.

Each right hand side reference of variable v in some program statement does not need to explicitly reference all of the bits in v during the execution of the statement. As a consequence, at each

point in v's live range, not all of the bits representing v are live. Therefore, different amounts of bits may be needed to hold the value of v in a register at different program points.

Definition 2. (Dead Bits) Given a variable v, which according to its declaration is represented by s bits, a subset of these s bits, say d, are *dead* at program point p if all computations following point p that use the current value of v can be performed without explicitly referring to the bits in d.

Definition 3. (Live Range Width) Given a program point p in variable v's live range, the *width of* v's *live range* at point p, denoted by $w_v(p)$, is defined such that the bits representing variable v according to its declaration can be divided into three contiguous sections as follows: a *leading section* of $l_v(p)$ *dead* bits; a *middle section* of $w_v(p)$ *live* bits; and a *trailing section* of $t_v(p)$ *dead* bits.



Let s_v denote a statement that refers to the value of variable v. We define $NOUSE(s_v, v)$ as an ordered pair (l, t) such that the leading l bits and trailing t bits of v need not be explicitly referred to during execution of s_v. The conditions under which only a subset of, and not all, bits of a variable v are sufficient for evaluating an expression are given in Fig. 2. The first three situations exploit the use of compile time constants in *left shift*, *right shift*, and *bitwise and* operations. The results computed by these expressions are only dependent upon subset of bits of v and thus the remaining bits are considered as not having been used. The next two situations exploit presence of zero bits in v. Leading zero bits present in v need not be explicitly held in a register to perform arithmetic and relational operations as the results of these operations can be correctly computed without explicitly referring to these bits. Similarly, the results of the *bitwise or* operation can be computed without explicitly referring to the leading and trailing zero bits of v. Therefore, we consider these zero bits as not having been used. Finally, in all other cases a right hand side reference to v is considered to use all bits of v, i.e. $NOUSE(s_v, v)$ is $(0, 0)$.

To identify dead bits and hence the width of the live range at each program point in the live range, we perform the following analysis. First we carry out *forward analysis* to compute a safe estimate of leading and trailing zero bit sections in each program variable at each program point. This information is needed in the computation of $NOUSE(s_v, v)$ in two of the cases described above (*arithmetic/relational* operations and *bitwise or* operation). Having computed $NOUSE(s_v, v)$ information fully, second we carry out *backward analysis* to identify the dead bit sections in each program variable at each program point. We describe these analyses next. Without loss of generailty, we assume in our discussion that all variables have the same declared bitwidth.

Leading and trailing zero bit sections

As described above, leading and trailing zeros need to be found because results of some operations can be computed without explicitly referring to them and thus these sections can be treated as dead bit sections. Forward analysis is employed to determine the leading and trailing zero bit sections for each variable, at each program point. When a variable v is being assigned, in some cases, by examining the expression on the right hand side we can determine the leading and trailing zero bit sections of v following the assignment. In case of a constant assignment $v = c$, by looking at the value of constant c, we can determine the zero bit sections of v. In case of signed numbers the leading zero bit section is formed by sign extension bits (i.e., from zeros or ones). A right (left) shift by a constant amount, i.e. $v = x >> c$ ($v = x << c$), results in the creation of leading (trailing) bit sections. A bitwise logical or (and) operation, i.e. $v = x \mid y$ ($v = x \ \& \ y$), results in propagation of zero bit sections. For a copy assignment $v = x$, the zero bit sections of x are simply propagated to v. If nothing can be asserted about the value being assigned to v, the analysis conservatively assumes that there are no leading or trailing zero bit sections.

Note that since zero bit sections of one variable may depend upon zero bit sections of another variable, all variables must be analyzed simultaneously. The meet operator for this forward analysis safely computes the smallest leading and trailing zero bit sections that are present in each variable across all incoming edges. The data flow equations for computing the zero bit sections are given in Fig. 3, where $ZBS_{in/out}[n, v]$ represents the zero bit sections of variable v at entry/exit of node n. Recall that for simplicity we only list the situations involving variables of the same bitwidth. When variables of different size are considered, additional opportunities arise. For example, when an unsigned short integer is assigned to an unsigned long integer, a leading zero bit section is created in the latter. More situations can be found to enhance the analysis.

Leading and trailing dead bit sections

Fig. 3 also gives the data flow equations for computing the dead bit sections. $DBS_{in/out}[n, v]$ represents the leading and trailing dead bit sections of variable v at entry/exit of node n. As expected, determination of dead bit sections is based upon backward analysis which examines each statement s_v to identify the subset of bits of variable v whose values are not used by the statement. This information is represented by $NOUSE(s_v, v)$ as described earlier, which can be computed for each statement given the results of the zero bit sections analysis. If the bit sections in $NOUSE(s_v, v)$ are dead at the point after statement s_v, then they are also dead immediately before statement s_v. If a statement defines v and does not use it, then all bits of the variable are dead which is indicated by $(\top, \top)$, i.e. leading and trailing dead bit sections of size equal to the width of the variable. The join operator conservatively computes those bit sections as dead at the exit of a node that are also dead at entry points of all successor nodes.

An example

The results of applying the above analyses are illustrated using an example shown in Fig. 4. For simplicity we use a straightline code example although our technique applies to programs with complex control flow structures. For the given code fragment, first we show the zero bit sections of each variable at the point it is assigned some value. Next we show the results of the dead bit sections analysis where the set of dead variables immediately following each statement are given. For example, immediately following statement 3 the higher order 4 bits of variable D are zero. In case the entire variable is dead we simply list the name of the variable (e.g., all of the involved variables are fully dead immediately preceding the code fragment). The results of the dead bit sections analysis are equivalent to the live ranges shown where the area enclosed in solid lines corresponds to the bit section that is not dead. If we examine the above ranges, it is easy to see that the maximum combined width of these live ranges at any program point is 32 bits. Therefore a *single* 32 bit register is sufficient to handle all these variables. Note that a traditional register allocator which ignores the widths of the variables will need *four* registers for this code fragment.

s_v	Characteristics of s_v	$NOUSE(s_v, v)$
$v >> t$	t is a compile time constant	$(0, t)$ - t trailing bits of v are not used.
$v << l$	l is a compile time constant	$(l, 0)$ - l leading bits of v are not used.
$v \ \& \ c$	c is a compile time constant with l leading and t trailing zero bits	(l, t) - l leading bits and t trailing bits of v are not used.
$v \ op \ \ldots$	op is an arithmetic or relational operator; v has at least l leading zero bits	$(l, 0)$ - l leading bits of v are not used.
$v \mid \ldots$	v has at least l leading zero bits and t trailing zero bits	(l, t) - l leading bits and t trailing bits of v are not used.
$f(v)$	other forms of statements that use v	$(0, 0)$ - all bits of v are used.

Figure 2: Partial use of a variable's bits.

Input: control flow graph $G = (N, E, start, end)$, where each node contains a single intermediate code statement.

definitions:

$(l_1, t_1) \wedge (l_2, t_2) = (min(l_1, l_2), min(t_1, t_2))$.
$(l_1, t_1) \vee (l_2, t_2) = (max(l_1, l_2), max(t_1, t_2))$.
$(l_1, t_1) \triangle (l_2, t_2) = (min(l_1, l_2), max(t_1, t_2))$.
$(l_1, t_1) \nabla (l_2, t_2) = (max(l_1, l_2), min(t_1, t_2))$.

boundary conditions: for each variable v, $ZBS_{in}[start, v] := DBS_{out}[end, v] := (\top, \top)$, where $\top$ is the bitwidth of variable v.

initialization: set all vectors to $(\top, \top)^S$, where S is the number of variables.

meet and join operators: $\bigwedge$ is the meet and join operator for the forward and backward analysis respectively.

Zero Bit Sections Analysis: Solve iteratively

$$ZBS_{in}[n, v] := \bigwedge_{p \in Pred(n)} (ZBS_{out}[p, v]).$$

$$ZBS_{out}[n, v] := \begin{cases} (l, t) & if\ n\ is\ v = c;\ c\ is\ a\ +ve\ constant;\ and \\ & (l, t)\ represents\ c's\ zero\ bit\ sections \\ (l + c, t - c) \triangle (\top, 0) & elseif\ n\ is\ v = x >> c;\ ZBS_{in}[n, x] = (l, t)\ and \\ & c\ is\ a\ +ve\ constant \\ (l - c, t + c) \nabla (0, \top) & elseif\ n\ is\ v = x << c;\ ZBS_{in}[n, x] = (l, t)\ and \\ & c\ is\ a\ +ve\ constant \\ ZBS_{in}[n, x] & elseif\ n\ is\ v = x \\ ZBS_{in}[n, x] \wedge ZBS_{in}[n, y] & elseif\ n\ is\ v = x \mid y \\ ZBS_{in}[n, x] \vee ZBS_{in}[n, y] & elseif\ n\ is\ v = x\ \&\ y \\ (0, 0) & elseif\ n\ defines\ v \\ ZBS_{in}[n, v] & otherwise \end{cases}$$

Dead Bit Sections Analysis: Solve iteratively

$$DBS_{in}[n, v] := \begin{cases} NOUSE(n, v) \wedge DBS_{out}[n, v] & if\ n\ uses\ v \\ (\top, \top) & elseif\ n\ defines\ v \\ DBS_{out}[n, v] & otherwise \end{cases}$$

$$DBS_{out}[n, v] := \bigwedge_{s \in Succ(n)} (DBS_{in}[s, v]).$$

Figure 3: Forward and backward bit sections analysis.

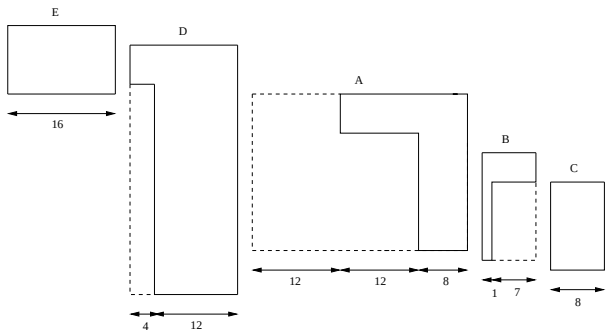
$int\ A;$ – 32 bits $short\ D, E;$ – 16 bits $char\ B, C;$ – 8 bits 1. $E = \ldots$ 2. $D = E + 1$ 3. $D = D >> 4$ 4. $A = (E << 4) \mid 0xf$ $this\ was\ E's\ last\ use.$ 5. $use\ A$ 6. $A = A >> 12$ 7. $B = \ldots$ 8. $C = (B \& 0x7f) + 1$ 9. $last\ use\ of\ A$ 10. $last\ use\ of\ B \& 0x80$ 11. $last\ use\ of\ C$ 12. $last\ use\ of\ D$	Zero Bit Sections 1. E: (0,0) 2. D: (0,0) 3. D: (4,0) 4. A: (12,0) 5. 6. A: (24,0) 7. B: (0,0) 8. C: (0,0) 9. 10. 11. 12.	Dead Bit Sections 0. A:B:C:D:E; $before\ 1$ 1. A:B:C:D 2. A:B:C 3. A:B:C:D(4,0) 4. A(12,0):B:C:D(4,0):E 5. A(12,0):B:C:D(4,0):E 6. A(24,0):B:C:D(4,0):E 7. A(24,0):C:D(4,0):E 8. A(24,0):B(0,7):D(4,0):E 9. A:B(0,7):D(4,0):E 10. A:B:D(4,0):E 11. A:B:C:D(4,0):E 12. A:B:C:D:E	

Figure 4: Illustration of live range construction.

1. $E = ...$ 2. $D = E + 1$ 3. $D = D >> 4$ 4. $A = (E << 4)\|0xf$ $;this\ was\ E's\ last\ use.$ 5. $use\ A$ 6. $A = A >> 12$ 7. $B = ...$ 8. $C = (B\&0x7f) + 1$ 9. $last\ use\ of\ A$ 10. $last\ use\ of\ B\&0x80$ 11. $last\ use\ of\ C$ 12. $last\ use\ of\ D$	1. $R_{0..15} = ...$ 2. $R_{16..31} = R_{0..15} + 1$ 3. $R_{16..27} = R_{20..31}$ $\boxed{R_{20..31} = R_{16..27}}$ 4. $R_{4..19} = R_{0..15}; R_{0..3} = 0xf$ 5. $use\ R_{0..19}$ 6. $R_{0..7} = R_{12..19}$ 7. $R_{8..15} = ...$ $\boxed{R_{16} = R_{15}}$ 8. $R_{8..15} = R_{8..14} + 1$ 9. $last\ use\ of\ R_{0..7}$ 10. $last\ use\ of\ R_{16}$ 11. $last\ use\ of\ R_{8..15}$ 12. $last\ use\ of\ R_{20..31}$	1. $R1_{0..15} = ...$ 2. $R1_{16..31} = R1_{0..15} + 1$ 3. $R1_{16..27} = R1_{20..31}$ 4. $R2_{4..19} = R1_{0..15}; R2_{0..3} = 0xf$ 5. $use\ R2_{0..19}$ 6. $R2_{0..7} = R2_{12..19}$ 7. $R2_{8..15} = ...$ 8. $R2_{16..23} = R2_{8..14} + 1$ 9. $last\ use\ of\ R2_{0..7}$ 10. $last\ use\ of\ R2_{15}$ 11. $last\ use\ of\ R2_{16..23}$ 12. $last\ use\ of\ R1_{16..27}$
Original code.	Code using one register.	Code using two registers.

Figure 5: Using registers with packed variables.

3. VARIABLE PACKING = ITERATIVE COALESCING OF INTERFERING NODES

In this section we present our variable packing algorithm. Let us first see the impact of variable packing on the generated code. Fig. 5 shows the code resulting after packing all variables of Fig. 4's example into *one register* R. The subscripts indicate the bit sections within R being referenced. It is clear that if bit section referencing is supported, a small number of registers can be used very effectively. Note that the shift operations of statements 3, 4, and 6 are translated into *intraregister bit section moves* which move a sequence of bits from one position to another. Also two additional intraregister moves, preceding statements 4 and 8, are required. These moves are required because sometimes when a variable that is allocated to the register is being defined, a free contiguous register bit section of the appropriate size may not be available. This is because the free bits may be *fragmented*. In this case the values of live variables present in the register must be shifted to combine the smaller free bit section fragments into one large contiguous bit section.

The algorithm we have developed sacrifices some of the variable packing opportunities in favor of fast execution time. For the preceding example, although one register is sufficient, our algorithms allocates A, B, and C to one register and D and E to another register. The resulting code based upon using two registers is shown in the figure. Since the variables are not packed as tightly, we find that there is no need to carry out the two intraregister moves for overcoming the problem of fragmentation of free bits.

Interference graph

Our approach to variable packing is to perform it as a *prepass* to global register allocation. The merit of this approach is that existing register allocation algorithms can be used without any modifications once variable packing has been performed. In addition, we design the variable packing algorithm to operate upon the live range *interference graph* which must be constructed any way to perform global register allocation. The nodes of an interference graph correspond to the live ranges. Interference edges are introduced between node pairs representing overlapping live ranges.

It is easy to see that from the perspective of the interference graph, variable packing can be performed through *iterative coalescing of interfering nodes*. In each step a pair of interfering live ranges can be coalesced into one node if no place in the program is their collective width greater than the number of bits in the register. After variable packing, register allocation is performed using the transformed interference graph.

Definition 4. (Maximum Interference Width) Given a pair of live ranges lr_1 and lr_2, the *maximum interference width* of these live ranges, denoted by $MIW(lr_1, lr_2)$, is the maximum combined width of these live ranges across all program points where the two live ranges overlap. Let $width(lr, p)$ denote the width of live range lr at program point p. $MIW(lr_1, lr_2)$ is computed as follows:

$$MIW(lr_1, lr_2) = width(lr_1, p) + width(lr_2, p)$$
$$iff\ \forall n\ st\ lr_1\ and\ lr_2\ overlap\ at\ n,$$
$$width(lr_1, n) + width(lr_2, n) \leq$$
$$width(lr_1, p) + width(lr_2, p).$$

It should be clear that (lr_1, lr_2) are coalesced *iff* $MIW(lr_1, lr_2) \leq |R|$, where $|R|$ is the number of bits in each register. We always assume that no variable has width greater than $|R|$.

The desired goal of coalescing can be set as achieving *maximal coalesing* that reduces the number of nodes in the interference graph to the mimimum possible that is achievable by any legal sequence of coalescing operations. However, the theorem we present next establishes that achieving maximal coalescing is NP-complete. In fact from our construction it can be seen that this result holds true even for straightline code.

Theorem (Live Range Coalescing is NP-complete).
Given a set of *live ranges* L, a constant $l < |L|$. Does there exist a live range coalesing that reduces the number of live ranges to l such that the width of no coalesced variable exceeds $|R|$ at any program point?

Proof. It is trivial to see that live range coalescing problem is in NP as given a solution it is easy to verify that it is correct in polynomial time. By performing a reduction from the *bin packing* problem (see [7], page 226) we can show that *live range coalescing* is NP-complete. The bin packing problem can be stated as follows.

Given a set of items U, a size $s(u)$ for each $u \in U$, and a positive integer *bin capacity* B. Is there a partition of U into disjoint sets $U_1, U_2, \cdots, U_K$ such that the sum of sizes of the items in each set U_i is B or less?

An instance of bin packing problem can be transformed into an instance of live range coalescing problem as follows. Corresponding to each item $u \in U$, we construct a live range of uniform width $s(u)$. We further assume that there is some program point where all live ranges fully overlap with each other. Now let $|R| = B$ and $l = K$. If we can find a live range coalescing that reduces the number of live ranges to l such that none of the coalesced live ranges has a width greater than $|R|$, we have essentially solved the corresponding instance of the bin packing problem. □

The overall outcome of coalescing depends upon the selection and order in which pairs of nodes are examined for coalescing. Given the above result we use an iterative coalescing heuristic which picks a node from the graph, coalesces it with as many neighboring nodes as possible, and then repeats this process for all remaining nodes. Let us briefly consider the runtime complexity of an iterative coalescing algorithm. The coalescing must have been carried out in a series of steps where in each step two nodes are coalesced. To determine whether two nodes, say lr_1 and lr_2, can be coalesced, we must check the condition $MIW(lr_1, lr_2) \leq |R|$ by scanning the two live ranges across the entire length of the program where the two live ranges overlap. The time complexity of this operation is $O(L)$, where L is bounded by the number of statements in the program. The number of coalescing operations is bounded by the number of nodes N in the interference graph. Thus the total time spent in coalescing is bounded by $O(N \times L)$.

To avoid the expensive operation of scanning two live ranges to compute $MIW(lr_1, lr_2)$ at the time of attempting coalescing, we explore the use of *fast* methods based upon the use of conservative estimates of $MIW(lr_1, lr_2)$. A conservative estimate can overestimate MIW but it must never underestimate it. Let us consider a simple and most obvious approximation. By scanning the entire program exactly once, we can precompute the maximum width of each live range lr, $MAX(lr)$. Using this information, *estimated maximum interference width* $EMIW(lr_1, lr_2)$ can be computed as follows: $EMIW(lr_1, lr_2) = MAX(lr_1) + MAX(lr_2)$ $(\geq MIW(lr_1, lr_2))$. Note we do not need to scan the program to compute $EMIW(lr_1, lr_2)$. While this method is simple and allows estimation of $EMIW$ of two live ranges at the time of iterative coalescing in $O(1)$ time, it fails to handle a common situation well. In Fig. 6 live ranges A and B are shown. It is clear that they can be allocated to a single 32 bit register. However, since $MAX(A) = MAX(B) = 32$ and therefore $EMIW(A, B) = 64$, we cannot coalesce them using this simple approach.

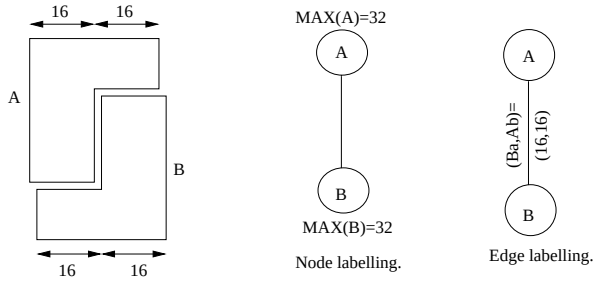


Figure 6: Node vs. edge labels.

To address the above problem with *node labels* we make use of *edge labels*. Each edge (A, B) is labelled with a pair of values (A_b, B_a), such that A_b and B_a represent the widths of A and B respectively at a program point corresponding to maximum interference width of A and B (i.e., $MIW(A, B) = A_b + B_a$). For the above example, the edge label is $(16, 16)$. The important observation is that by looking at the edge label we can now determine that coalescing of A and B is possible because their combined width at any program point does not exceed 32 (because $16 + 16 = 32$). The edge labels are more formally defined below.

Definition 5. (Interference Graph Labels) Initially each interference edge (A, B) is labelled with (A_b, B_a) where A_b and B_a are the contributions of A and B to $MIW(A, B)$ (i.e., $MIW(A, B) = A_b + B_a$). Subsequently, each edge (C, D) formed after coalescing, is labelled with (C_d, D_c) where C_d and D_c are the contributions of C and D to $EMIW(C, D)$ (i.e., $EMIW(C, D) = C_d + D_c$).

When nodes are coalesced, labels for the edges emanating from the newly created node must be computed. It is during this process that some imprecision is introduced. We have developed a *fast* and *highly accurate* method for computing the edge labels. Next we present this method in detail.

Updating edge labels following coalescing

If there was an edge between a node C and one or both of nodes A and B, then there will be an edge between AB and C in the transformed graph. We must determine the label (AB_c, C_{ab}) for this edge. Two cases that arise are handled as shown in Fig. 7.

The first case involves the situation in which C was connected by an edge to either A or B. In this case the label of edge (AB, C) will be same as the label on the edge (A, C) or (B, C) as the case may be. Since C interferes only with A (or B), after coalescing of A and B the maximum interference width between AB and C is same as maximum interference width between A (or B) and C. It should be noted that no additional imprecision is introduced during the generation of the label for edge (AB, C).

The second case considers the situation in which there is an edge between C and both A and B. In this case additional imprecision may be introduced during the estimation of (AB_c, C_{ab}) for edge (AB, C) as this label is based upon a conservative estimate of $EMIW(A, B, C)$. Our goal is to carry out this estimation quickly by avoiding examining the complete live ranges corresponding to nodes A, B and C. In addition, we would like to obtain a label that is as precise as possible based upon the existing labels of the three nodes and edges between them.

Three candidate estimates for $EMIW(A, B, C)$ denoted by E_A, E_B, and E_C in Fig. 7 are considered. E_A is the estimate of the sum of widths of A, B, and C at a point where maximum interference width between B and C takes place. At such a point, the best estimates for the widths of A, B and C are $max(A_b, A_c)$, B_c and C_b respectively (i.e., $E_A = max(A_b, A_c) + B_c + C_b$). Similarly E_B (E_C) represents the point at which maximum interference width of A and C (A and B) takes place. Therefore values of E_B and E_C can be similarly computed. While it may not be the case that $EMIW(A, B, C)$ is equal to any of the three computed values (i.e., E_A, E_B and E_C) we can derive a conservative estimate of $EMIW(A, B, C)$ from these values. In particular, if we sort the values of E_A, E_B, and E_C, the *intermediate value* E_{int} is a safe approximation for $EMIW(A, B, C)$. Therefore as shown in Fig. 7, we choose this value and depending upon whether E_{int} is E_A, E_B or E_C, we accordingly compute (AB_c, C_{ab}). The theorem in Fig. 8 formally proves the correctness of this method.

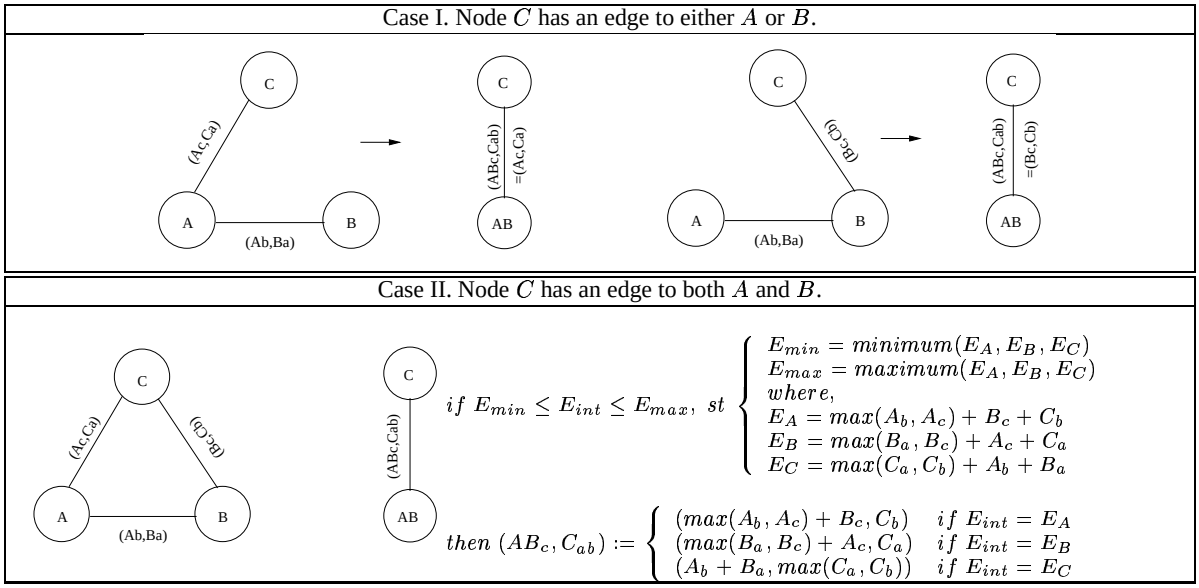


Figure 7: Updating labels after coalescing A and B.

From the *intermediate value theorem* it follows that a single coalescing operation takes $O(1)$ time. Each coalescing operation removes a node from the interference graph. Therefore the number of coalescing operations is bounded by the number of nodes N in the interference graph. Hence the run time complexity of the coalescing operations is bounded by $O(N)$. Recall that the slow algorithm had a complexity of $O(N \times L)$.

Example

Let us apply the coalescing operations using the *intermediate value theorem* to the interference graph of the live ranges constructed for an example in Fig. 4. We assume that the registers are 32 bits wide for this example. From the live ranges constructed we first build the five node interference graph shown in Fig. 9.

While the nodes in the interference graph can be coalesced in a number of ways, one such order is shown in Fig. 9. First we merge D and E. According to rules for handling *Case I*, the labels for all edges emanating from D become labels of the corresponding edges emanating from DE. In the next two steps nodes A, B, and C are coalesced during which *Case II* arises. Therefore, the labels on edges are updated using the intermediate value theorem giving the results shown in the figure.

Note that if the bitwidths of the variables are ignored, the original interference graph requires 4 registers to color as the graph contains a clique of four nodes. In contrast a register allocator will need to use two colors to color the coalesced interference graph. Thus the proposed coalescing algorithm reduces the register requirements for the interference graph from 4 registers to 2 registers. The code based upon usage of two registers was shown in Fig. 5.

Now let us conisder the result of application of simple coalescing approach which only maintains node labels. Assuming that the same pairs of nodes are considered for coalescing as were considered during the application of algorithm based upon edge labels in Fig. 9, we can perform at most two coalescing operations as shown in Fig. 10. Thus, in this case 3 registers would be required. Therefore using edge labels is superior to using node labels in this example.

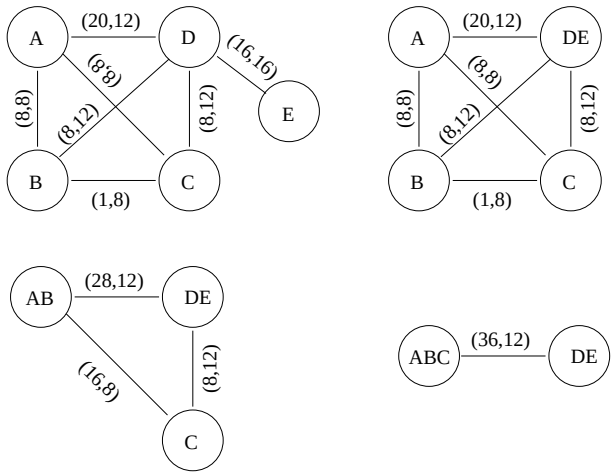


Figure 9: Illustration of node coalescing.

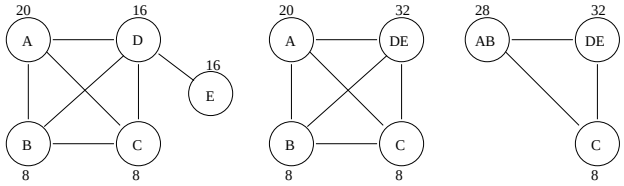


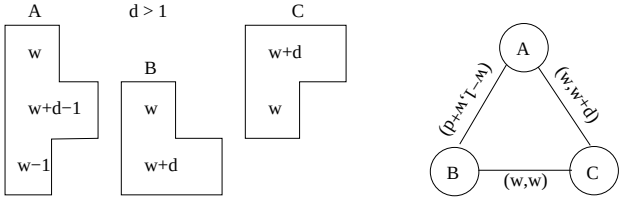
Figure 10: Node coalescing using node labels.

Theorem (Intermediate Value Theorem).

$if\ E_{min} \leq E_{int} \leq E_{max},\ st\ \left\{ \begin{array}{l} E_{min} = minimum(E_A, E_B, E_C) \\ E_{max} = maximum(E_A, E_B, E_C) \end{array} \right.\ ,\ EMIW(A,B,C) = E_{int}$ is safe.

Proof. The proof is carried out in two parts. Lemma 1 shows that in general E_{min} is not a safe estimate for $MIW(A,B,C)$ because E_{min} can be less than $MIW(A,B,C)$. Lemma 2 shows that if E_{min} is less than $MIW(A,B,C)$, then values of both E_{int} and E_{max} are greater than $MIW(A,B,C)$. From Lemma 1 and Lemma 2 it follows that E_{int} is the best safe estimate for $MIW(A,B,C)$ from among the three values, E_A, E_B and E_C.

(Lemma 1) $E_{min} < MIW(A,B,C)$ can be true: Consider the construction of live ranges as shown in the figure below. Note that $d > 1$ in this construction which clearly confirms that indeed E_{min} may not be a safe estimate of $MIW(A,B,C)$ as $E_{min} = E_A < MIW(A,B,C)$.



$MIW(A,B,C) = 3w + d - 1$
$E_A = max(w-1, w) + w + w = 3w$
$E_B = max(w+d, w) + w + w + d = 3w + 2d$
$E_C = max(w+d, w) + w - 1 + w + d = 3w + 2d - 1$
$\Rightarrow E_{min} = E_A < MIW(A,B,C).$

(Lemma 2) $E_{min} < MIW(A,B,C) \Rightarrow E_{max} \geq E_{int} \geq MIW(A,B,C)$: Let $MIW(A,B,C) = W_A + W_B + W_C$, where W_A, W_B, and W_C are contributions of A, B and C to $MIW(A,B,C)$. By definition of MIW it must be the case that:

⓪ $A_b + B_a \geq W_A + W_B$,
① $A_c + C_a \geq W_A + W_C$, and
② $B_c + C_b \geq W_B + W_C$.

Without any loss of generality, let us assume that $E_{min} = E_A$. Given that $E_{min} < MIW(A,B,C)$ it follows that:

$E_A = max(A_b, A_c) + B_c + C_b < W_A + W_B + W_C$ or
③ $W_A + W_B + W_C > max(A_b, A_c) + B_c + C_b$.

$[W_A + W_B + W_C > max(A_b, A_c) + B_c + C_b]$ ③ and $[B_c + C_b \geq W_B + W_C]$ ②
$\Rightarrow$ $W_A > max(A_b, A_c)$
$\Rightarrow$ ④ $W_A > A_b$ and
⑤ $W_A > A_c$.

$[A_b + B_a \geq W_A + W_B]$ ⓪ and $[W_A > A_b]$ ④ $\Rightarrow$ $B_a > W_B$ $\Rightarrow$ ⑥ $max(B_a, B_c) > W_B$
$[A_c + C_a \geq W_A + W_C]$ ① and $[W_A > A_c]$ ⑤ $\Rightarrow$ $C_a > W_C$ $\Rightarrow$ ⑦ $max(C_a, C_b) > W_C$

$[max(B_a, B_c) > W_B]$ ⑥ and $[A_c + C_a \geq W_A + W_C]$ ①
$\Rightarrow$ $max(B_a, B_c) + A_c + C_a > W_A + W_B + W_C$
$\Rightarrow$ ⑧ $E_B > MIW(A,B,C)$.

$[max(C_a, C_b) > W_C]$ ⑦ and $[A_b + B_a \geq W_A + W_B]$ ⓪
$\Rightarrow$ $max(C_a, C_b) + A_b + B_a > W_A + W_B + W_C$
$\Rightarrow$ ⑨ $E_C > MIW(A,B,C)$.

We have shown that if $E_A < MIW(A,B,C)$, then $E_B > MIW(A,B,C)$ and $E_C > MIW(A,B,C)$. Given that E_A is E_{min}, $E_{int} = min(E_B, E_C)$ and $E_{max} = max(E_B, E_C)$. Thus we conclude that:
$E_{min} < MIW(A,B,C) \Rightarrow E_{max} \geq E_{int} \geq MIW(A,B,C)$.

From Lemma 1 and Lemma 2 it follows that $EMIW(A,B,C) = E_{int}$ is the best safe estimate for $MIW(A,B,C)$ from among the three values, E_A, E_B and E_C. Hence the proof of the *intermediate value theorem* is complete. □

Figure 8: The intermediate value theorem.

Priority based coalescing

So far we have focussed on the fundamental issues of bitwidth aware register allocation (i.e., analysis for live range construction and variable packing using node coalescing). We have not addressed the following issues: Is coalescing always good? In what order should node coalescing be attempted?

While coalescing can reduce the chromatic number of a graph, this in not always the case. In some situations coalescing may increase the chromatic number of graph – for the graph shown below the chromatic number is two before coalescing but it increases to three after coalescing. A solution for preventing harmful coalescing was proposed by Briggs et al. [1]. They observed that if the node created by coalescing of two nodes has fewer than k neighbors with degree of k or more, where k is the number of colors, the resulting node will always be colorable. Thus, they propose restricting coalescing to situations where resulting nodes are guaranteed to be colorable.

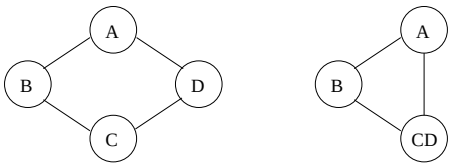


Figure 11: Increase in chromatic number due to coalescing.

The order in which node coalescings are attempted impacts the shape of the final graph and thus the number of colors required to color the resulting graph. For example, if we reconsider the example of Fig. 10 and merge nodes A and B as well as nodes C and D, the resulting graph contains three nodes which can be colored using two colors as opposed to three colors that are required by resulting graph of Fig. 10.

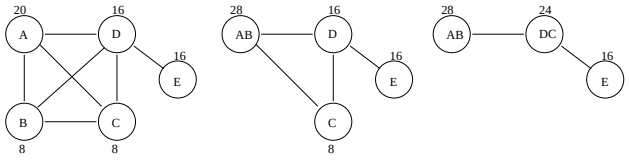


Figure 12: The impact of ordering of coalescing operations.

One approach that we propose to address the above problem is to assign priorities to all of the nodes. Node with the highest priority, say n, is picked and neighbors of n are considered for coalescing in decreasing order of priority. When no more nodes can be coalesced with n, the next highest priority node is picked and the above process repeated. The priority of a live range l is computed as shown below. The greater the savings due to elimination of loads and stores, the higher is the priority. However, the savings are normalized with respect to the the amount of register resources used. The register usage is based upon the duration and the number of bits that are occupied by the live range. Hence, it is simply the area of the live range which can be obtained by summing together the number of bits occupied by the live range at all relevant program points.

$$Priority(lr) \propto \frac{Savings(lr)}{RegisterUsage(lr)}$$

$$= \frac{Loads\&StoresAvoided}{LiveRangeArea}$$

$$= \frac{Loads\&StoresAvoided}{\sum_{\forall p} width(lr,p)}.$$

Our algorithm for carrying out node coalescing followed by register allocation is summarized in Fig. 13. Following the iterative node coalescing phase each set of coalesced variables is given a new name and the code is transformed to use this name. In addition, intravariable moves are introduced to preserve program semantics. The resulting interference graph is then processed using a traditional coloring based register allocator.

1. Construct *interference graph*.
2. Label edges with *interference widths*.
3. Construct *prioritized node list*.
4. **while** node list $\neq \phi$ **do**
5. Get a node, say n, from prioritized node list.
6. **for** each node a in n's *adjacency list* **do**
7. *Attempt coalescing* a with n.
8. **if** successful, *update* graph and prioritized list.
9. **endfor**
10. **endwhile**
11. Replace each coalesced variable set with a new name.
12. Introduce *intravariable moves*.
13. Perform *coloring* based register allocation.

Figure 13: Algorithm summary.

4. EXPERIMENTAL RESULTS

We evaluated the proposed technique using benchmarks taken from the `Mediabench` [12] (`adpcm` and `g721`), `NetBench` [14] (`crc` and `dh`), and `Bitwise` project at MIT [17] (`SoftFloat`, `NewLife`, `MotionTest`, `Bubblesort` and `Histogram`) as they are representative of a class of applications important for the embedded domain. We also added an image processing application (`thres`). We applied our technique to selected functions from these benchmarks that are large in size.

We constructed the interference graphs for the selected functions and measured the register requirements for fully coloring these graphs using the following algorithms: (a) *Bitwidth unaware* algorithm which at any given time allows only a single variable to reside in a register; (b) *Naive coalescing* (NC) algorithm that labels each node with its declared width and uses these labels to perform coalescing; and (c) *Our coalescing* (OC) algorithm that builds live ranges using bit section analysis and labels edges with maximum interference width information to drive coalescing. In all three cases the register requirements were computed by repeatedly applying Chaitin's algorithm to find the minimum number of registers for which the graph could be fully colored.

The results of our experiments are given in Table 1. While the OC algorithm reduces register requirements by 10% to 50%, NC algorithm is nearly not as successful. By reducing the register requirements by a few registers, the quality of code can be expected to improve significantly. This is particularly true for the ARM processor with bit section referencing extensions [13] in context of which this research is being carried out as ARM has 16 only registers.

Table 1: Register requirements.

Benchmark Function	Registers Used		
	BU	NC	OC
adpcm_decoder	15	15	13
adpcm_coder	18	18	15
g721.update	15	12	10
g721.fmult	4	3	3
g721.quantize	6	5	5
thres.memo	6	6	4
thres.coalesce	10	10	5
thres.homogen	11	11	6
thres.clip	6	6	4
SoftFloat.mul32To64	8	7	7
NewLife.main	7	7	4
MotionTest.main	6	6	5
Bubblesort.main	9	9	7
Histogram.main	7	7	6
crc.main	10	10	9
dh.encodelastquantum	7	7	4

Table 2: Benefits of coalescing.

Benchmark Function	Number of Nodes	
	Before	After
adpcm_decoder	17	15
adpcm_coder	20	17
g721.update	22	15
g721.fmult	8	7
g721.quantize	8	6
thres.memo	6	4
thres.coalesce	10	5
thres.homogen	12	7
thres.clip	7	5
SoftFloat.mul32To64	14	12
NewLife.main	8	5
MotionTest.main	9	7
Bubblesort.main	15	11
Histogram.main	13	11
crc.main	12	11
dh.encodelastquantum	8	5

Table 3: Change in maximum clique size.

Benchmark Function	Number of Nodes	
	Before	After
adpcm_decoder	15	13
adpcm_coder	18	15
g721.fmult	4	3
g721.quantize	6	5
thres.memo	6	4
thres.coalesce	10	5
thres.homogen	11	6
thres.clip	6	4
NewLife.main	7	4
MotionTest.main	6	5
crc.main	10	9
dh.encodelastquantum	7	4

Table 4: Live ranges with bitwidth < 32 bits.

Number of live ranges	Live Range Widths (bits)	
	Declared Size	Max Size After BSA
adpcm.decoder		
1	32	4
2	32	1
adpcm.coder		
2	32	1
1	32	8
1	32	3
g721.update		
7	16	16
1	32	3
4	16	5
1	16	1
1	8	1
g721.fmult		
3	16	16
1	16	12
2	16	4
g721.quantize		
3	16	16
1	16	12
2	16	4
thres.memo		
2	32	10
1	32	2
thres.coalesce		
2	32	10
5	32	8
thres.homogen		
2	32	10
5	32	8
1	32	2
thres.clip		
2	32	10
1	32	8
SoftFloat.mul32To64		
2	32	1
4	16	16
NewLife.main		
1	32	12
2	32	6
2	32	4
MotionTest.main		
1	32	16
1	32	6
1	32	4
Bubblesort.main		
3	32	16
6	32	10
Histogram.main		
1	32	16
1	32	12
2	32	10
1	32	8
crc.main		
1	64	56
1	32	8
dh.encodelastquantum		
4	32	6

To further understand the significance of the two key steps of our algorithm, namely live range construction based upon bit section analysis and node coalescing, we examined the data in greater detail. The results in Table 2 show the extent to which node coalescing reduces the number of nodes in each interference graph. As we can see significant amount of coalescing is observed to occur. The data in Table 4 shows the significance of our live range construction algorithm. The *declared* widths of live ranges as well as their reduced *maximum widths* after bit section analysis (BSA) are given. As we can see, for many live ranges, the declared widths are much larger than their reduced maximum widths. The reason why NC algorithm is nearly not as successful as our OC algorithm is made clear in part by this data – the declared bitwidths of variables are often much greater than their true bitwidths.

Finally it should be noted that although coalescing does not necessarily guarantee a reduction in register requirements, in most of the programs a significant reduction was observed. We looked at the interference graphs to understand why this was the case. We found that in most of these programs there were significantly large cliques present which accounted for most of the register requirements. Moreover the maximum sized clique in the interference graph typically contained multiple subword data items. Thus, node coalescing resulted in a reduction in the size of the maximum sized clique and hence the register requirements. Table 3 shows the reduction in the size of the largest cliques for the programs where the above observation holds. The benchmarks which did not exhibit this behavior are omitted from the table. In the case of `Soft-Float` and `Bubblesort` there were no large cliques while in the case of `g721.update` and `Histogram` although large cliques were present, they were not reduced in size by node coalescing.

5. RELATED WORK

Bit Section Analysis

Stephenson et al. [17] proposed *bitwidth analysis* to discover narrow width data by performing value range analysis. Once the compiler has proven that certain data items do not require a complete word of memory, they are compressed to smaller size (e.g., word data may be compressed to half-word or byte data). There are a number of important differences between bitwidth analysis and our analysis for live range construction. First our analysis is aimed at narrowing the width of a variable at each program point as much as possible since we can allocate varying number of register bits to the variable at different program points. Second, while our approach can eliminate a trailing bit section, value range analysis can never do so. Our approach can eliminate a leading bit section of dead bits which contains non-zero values while value range analysis can only eliminate a leading bit section if it contains zero bits through out the program. Budiu et al. [2] propose a analysis for inferring the values of individual bits. This analysis is much more expensive than our analysis as it must analyze each bit in the variable while our approach maintains summary information in form of three bit sections for each variable. Finally, the analysis by Zhang et al. [9] is aimed at automatic discovery of packed variables, while this paper is aimed at carrying out analysis to facilitate variable packing.

Memory coalescing and data compression

Davidson and Jinturkar [4] were first to propose a compiler optimization that exploits narrow width data. They proposed *memory coalescing* for improving the cache performance of a program. Zhang and Gupta [20] have proposed techniques for *compressing* narrow width and pointer data for improving cache performance. However, both of these techniques were explored in context of general purpose processors. Therefore aggressive packing of scalar variables into registers was not studied. In contrast, the work we present in this paper is aimed at new class of embedded processors where efficient use of small number of registers is made possible by holding multiple values in a single register. The only work we are aware of that deals with register allocation for processors that support bit section referencing is by Wagner and Leupers [18]. However, their work exploits bit section referencing in context of variables that already contained packed data. They do not carry out any additional variable packing as described in this paper. Some multimedia instruction sets support long registers which can hold multiple words of data for carrying out SIMD operations [5, 11]. Compiler techniques allocate array sections to these registers. In contrast, our work is aimed at shrinking scalars to subword entities and packing them into registers which are one word long. The scalar variables that we handle are ignored by superword techniques. Finally in context of embedded processors work has been done on dealing with irregular constraints on register allocation (e.g., [10]). However, our work is being done in context of ARM instruction set with bit referencing extensions where bit section packing is an important issue [13].

6. CONCLUSIONS

Multimedia and network processing applications make extensive use of subword data. Moreover embedded processors typically support a small number of word sized registers. Instruction set support for bit section referencing provides us with an opportunity to make effective use of small number of registers by packing multiple subword sized variables into a single register, without incurring any additional penalty for accessing packed variables. However, no techniques exist for either identifying variable bitwidth data or packing them into registers. We presented the first algorithms to solve both of these problems. We presented efficient analyses for constructing variable bitwidth live ranges and an efficient variable packing algorithm that operates on an enhanced interference graph. Our experiments show that the proposed techniques can reduce register requirements of embedded applications by 10% to 50%.

7. ACKNOWLEDGEMENTS

This work is supported by DARPA award F29601-00-1-0183 and National Science Foundation grants CCR-0208756, CCR-0105535, CCR-0096122, and EIA-9806525 to the University of Arizona.

8. REFERENCES

[1] P. Briggs, K.D. Cooper, and L. Torczon, "Improvements to graph coloring register allocation," *ACM Transactions on Programming Languages and Systems* (TOPLAS), 16(3):428-455, May 1994.

[2] M. Budiu, M. Sakr, K. Walker, and S.C. Goldstein, "BitValue Inference: Detecting and Exploiting Narrow Width Computations," *6th European Conference on Parallel Computing* (Euro-Par), August 2000.

[3] G. J. Chaitin, "Register allocation and spilling via graph coloring," *SIGPLAN Symposium on Compiler Construction*, June 1982.

[4] J. Davidson and S. Jinturkar, "Memory access coalescing : a technique for eliminating redundant memory accesses," *ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI), pages 186–195, 1994.

[5] R.J. Fisher and H.G. Dietz, "Compiling for SIMD within Register," *11th International Workshop on Languages and Compilers for Parallel Computing* (LCPC), *LNCS, Springer Verlag*, Chapel Hill, NC, August 1998.

[6] J. Fridman, "Data Alignment for Sub-Word Parallelism in DSP," *IEEE Workshop on Signal Processing Systems* (SiPS), pages 251-260, 1999.

[7] M.R. Garey and D.S. Johnson, *Computers and intractability: a guide to the theory of NP-completeness*, 1979.

[8] L. George and A.W. Appel, "Iterated register coalescing," *ACM Transactions on Programming Languages and Systems* (TOPLAS), 18(3):300-324, May 1996.

[9] R. Gupta, E. Mehofer, and Y. Zhang, "A Representation for Bit Section Based Analysis and Optimization," *International Conference on Compiler Construction* (CC), LNCS 2304, Springer Verlag, pages 62-77, Grenoble, France, April 2002.

[10] A. Koseki, H. Komatsu, and T. Nakatani, "Preference-Directed Graph Coloring," *ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI), pages 33–44, June 2002.

[11] S. Larsen and S. Amarasinghe, "Exploiting Superword Level Parallelism with Multimedia Instruction Sets," *ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI), pages 145–156, Vancouver B.C., Canada, June 2000.

[12] C. Lee, M. Potkonjak, and W.H. Mangione-Smith, "Mediabench: a tool for evaluating and synthesizing multimedia and communications systems," *IEEE/ACM International Symposium on Microarchitecture* (MICRO), Research Triangle Park, North Carolina, December 1997.

[13] B. Li and R. Gupta, "Bit section instruction set extension of ARM for embedded applications," *International Conference on Compilers, Architecture, and Synthesis of Embedded Systems* (CASES), Grenoble, Oct. 2002.

[14] G. Memik, B. Mangione-Smith, and W. Hu, "NetBench: A Benchmarking Suite for Network Processors," *IEEE International Conference Computer-Aided Design* (ICCAD), Nov. 2001.

[15] X. Nie, L. Gazsi, F. Engel, and G. Fettweis, "A New Network Processor Architecture for High Speed Communications," *IEEE Workshop on Signal Processing Systems* (SiPS), pages 548-557, 1999.

[16] D. Seal, Editor, "ARM architecture reference manual," Second Addition, Addison-Wesley.

[17] M. Stephenson, J. Babb, and S. Amarasinghe, "Bitwidth Analysis with Application to Silicon Compilation," *ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI), pages 108–120, Vancouver B.C., Canada, June 2000.

[18] J. Wagner and R. Leupers, "C Compiler Design for an Industrial Network Processor," *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems* (LCTES), pages 155-164, June 2001.

[19] T. Wolf and M. Franklin, "Commbench - a telecommunications benchmark for network processors," *IEEE International Symposium on Performance Analysis of Systems and Software* (ISPASS), April 2000.

[20] Y. Zhang and R. Gupta, "Data Compression Transformations for Dynamically Allocated Data Structures," *International Conference on Compiler Construction* (CC), LNCS 2304, Springer Verlag, pages 14-28, Grenoble, France, April 2002.