Speculative Subword Register Allocation in Embedded Processors*

Bengu Li[1], Youtao Zhang[2], and Rajiv Gupta[1]

The Univ. of Arizona, Dept. of Computer Science, Tucson, Arizona[1]
The Univ. of Texas at Dallas, Dept. of Computer Science, Richardson, Texas[2]

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. We propose an instruction set extension to the ARM embedded processor which allows two data items to reside in a register as long as each of them can be stored in 16 bits. The instructions are used by the register allocator to speculatively move the value of an otherwise spilled variable into a register which has already been assigned to another variable. The move is speculative because it only succeeds if the two values (value already present in the register and the value being moved into the register) can be simultaneously held in the register using 16 bits each. When this value is reloaded for further use, an attempt is first made to retrieve the value from its speculatively assigned register. If this attempt succeeds, load from memory is avoided. On an average our technique avoids 47% of dynamic reloads caused by spills.

1 Introduction

Recent research has shown that embedded applications, including network and media processing applications, contain significant amounts of narrow width data [4, 13]. Network processing applications usually pack the data before transmission and unpack it before processing. The unpacked data is composed of subword data. Media applications typically process byte streams. Since subword data requires fewer bits to store than the machine word size, its presence can be exploited by many optimizations that can help satisfy the tight area, performance and power constraints of embedded systems. Recent work has focused on exploiting narrow width data to carry out memory related optimizations [5, 14–16]. In this paper we show how narrow width data can be exploited to make effective use of small number of registers provided by embedded processors. We address this problem in context of the ARM processor [10].

The main objective of this work is to develop the architectural and compiler support through which the registers can be used more effectively in presence of subword data. In particular, our goal is to allow two variables to be simultaneously assigned to the same register such that if their values can be represented

* Supported by Microsoft, Intel, and NSF grants CCR-0324969, CCR-0220334, CCR-0208756, CCR-0105355, and EIA-0080123 to the Univ. of Arizona.

using 16 bits, they can be simultaneously held in a register. To demonstrate that this objective is worth pursuing we collected some data by executing a set of embedded applications. We found that the average bitwidth across all variables over entire program execution ranges from 15.87 to 20.74 bits across the benchmarks. We also found that the percentage of variables whose average width for the entire execution does not exceed 16 bits ranges from 33% to 61% across the benchmarks. Thus, it is quite clear that many variables can be stored using 16 bits for significant periods of time during execution.

Table 1. Dynamically observed narrow width data.

Benchmark	Dynamic Bitwidth	Variables $\leq$ 16 Bits
g721.decode	17.43	49%
g721.encode	18.99	44%
epic	16.35	61%
unepic	19.52	45%
mpeg2.decode	17.59	45%
mpeg2.encode	15.87	59%
adpcm.dec	17.07	50%
adpcm.enc	16.87	54%
drr	18.81	35%
frag	20.52	33%
reed	20.74	38%
rtr	19.03	38%

Generally the observed subword data can be divided into two categories: (static) a variable is declared as a word but in reality the values assigned to the variable can never exceed 16 bits; and (dynamic) a variable is declared as a word but in practice the values assigned to it do not exceed 16 bits most of the time during a program run. In our prior work on bitwidth aware register allocation, the compiler is responsible for identifying the upperbound on the bitwidth of variables and then multiple subword variables are packed into individual registers during register allocation [11, 6]. This approach exploits only the opportunities due to static subword data.

In this paper, we propose a *speculative subword register allocation* (SSRA) scheme. A *speculative* allocation pass is introduced after the *normal* register allocation pass. In the speculative pass, an additional variable may be allocated to a register which was assigned to another variable in the normal pass. A decision to assign another variable to an already allocated register is made only if the value profile data indicates that *most of the time* the two variables can fit into a single register. While the value of variable assigned to the register in the normal pass is guaranteed to be present in the register, the value of the variable assigned to the same register in the speculative pass may or may not be present in the register depending upon whether the two values can fit into a single register or not. Code is generated in a way that while loads are generated to reload the value of the variable assigned a register during the speculative pass, these loads are only executed if the value cannot be saved in the register. While the benefit of this approach is the reduction in the number of dynamically executed loads, in case the value cannot be kept in the register we pay an extra cost. This extra cost is due to additional instructions introduced to save and retrieve

speculatively stored values in registers. The main advantage of this approach is that both categories of subword data (static and dynamic) can be exploited. Moreover, static bitwidth analysis [9, 11, 3] that is otherwise used to compute bitwidths of variables in no longer needed. Instead, speculative register allocation is driven by value profiles. It should be noted that our work is orthogonal to speculative register promotion technique of [7]. While speculative register promotion allows allocation of registers to variables in presence of aliasing, our work assigns already allocated registers to additional variables when no free registers are available.

The rest of the paper is organized as follows. In section 2 we discuss bitwidth aware register allocation and identify the challenges in developing a speculative register allocation scheme. The architectural support is introduced in section 3. Section 4 discusses the speculative register allocation algorithm. The implementation and experimental results are presented in section 5. Related work and conclusions are given in sections 6 and 7.

2 Bitwidth Aware Register Allocation: Static vs. Dynamic Narrow Width Data

The discussion of register allocation in this paper targets embedded processors where small modifications can be made to effectively manipulate subword data. Each register R can contain up to two entities that are of same size, i.e. it can hold a whole word $R_{1..32}$ or two half-words $R_{1..16}$ and $R_{17..32}$.

Let us consider the existing approach to bitwidth aware register allocation [11]. Figure 1(a) contains a code fragment in which variables b and c are assigned to registers $R1$ and $R2$ respectively. Moreover there is no free register for variable a. Thus, spill code is generated to store new value of a back to memory after computation and a load is introduced to bring the value from memory before a is used (Figure 1(b)). The approach in [11] determines the bitwidths of the variables using data flow analysis. Let us assume that the resulting bitwidths indicate that all three variables can be stored in 16 bits. Thus, the variables can each be assigned to use half of a register as shown in Figure 1(c). In the transformed code, instead of being spilled to memory, variable a is assigned to use half of a register $R1_{17..32}$ while the other half of the same register is used to hold the value of variable b.

(a) original C code

```
/* b can always be
represented using 16
bits */

a = b + 1;
…

c = a - 10;
```

(b) with spill code

```
; b → R1
; c → R2

add R3, R1, 1
sw R3, addr_a
…
lw R4, addr_a
sub R2, R4, 10
```

(c) after bitwidth aware RA

```
; b → R1_{1..16}
; c → R2_{1..16}
; a → R2_{17..32}

add R2_{17..32} , R1_{1..16} , 1
…

sub R2_{1..16} , R2_{17..32} ,10
```

Fig. 1. Bitwidth aware register allocation.

While the above approach is effective, there are situations under which opportunities for subword register allocation cannot be detected and/or exploited. It is possible that the compiler is unable to establish that variables a, b and

c occupy only half a word or that they occupy half a word most of the time and only rarely take up more space. In particular, the three situations under which the above approach fails are as follows: (a) The program does not contain enough information to enable discovery of true bitwidths. For example, an input variable may be declared as a full word variable although the valid inputs do not exceed 16 bits. The compiler will fail to realize that the input variable represents subword data; in addition, bitwidths of variables that depend upon this input will also be likely overestimated; (b) Even if the program contains information to infer the true bitwidths of variables, the imprecision in static analysis may lead us to conclude that the bitwidths exceed 16 bits; and (c) We miss the opportunity to optimize variables that represent subword data most of the time but only sometimes exceed 16 bits. For example, for the code in Figure 3(a) the value profiles may show that for most executions or for most of the time during a single execution variables a, b, and c take values that can be represented in 16 bits. However, existing register allocation techniques cannot exploit this information and will introduce spill code as in Figure 1(b).

In summary we can say that while the existing method for bitwidth aware register allocation can take advantage of statically known subword data, it cannot take advantage of dynamically observed subword data. In this paper we address this issue by proposing a speculative mechanism for packing two variables into a single register. Profiling information is used to identify variables which when packed together are highly likely to always have their values fit into a single register. If the values do fit into the same register, loads associated with reloading of values are avoided; otherwise they are executed. While the basic idea of our approach is clear, there are architectural and compiler challenges to achieving such a solution which are as follows:

Architectural support. The key challenge here is to design instruction set extensions through which the mechanism for speculative packing of two variables into one register can be exposed to the compiler. Compiler should be able to control which variable is *guaranteed* to be found in a register and which variable is *expected* to be present. Instruction for speculatively storing a value into a register and *checking* whether it is present are needed. Finally once two values are present in a register we need to be able to *address* them.

Compiler support. We need to develop an algorithm for global register allocation such that it leads to performance improvements when speculative register assignments are made. This requires that we carefully choose the pair of variables that are assigned to the same register. The compiler must choose the variable whose value is guaranteed to be found in the register and one whose value is expected to be found in the register. Since there is extra cost due to checking whether a speculatively stored value is present in a register, the compiler must use profile data to pair variables such that it is highly likely that the values of both variables will be able to reside simultaneously in the same register. Finally speculative register assignment should be carefully integrated into conventional global register allocation algorithm.

3 Architectural Support

In this section, we discuss the necessary architectural support to enable speculative subword register allocation. To support subword accesses, we attach one bit with each register, allocate a global bit in status word register and add four new instructions. These extensions are described in greater detail next.

3.1 Register File Enhancement

The register file in ARM processors includes 16 user visible 32-bit registers. We add one extra bit B to each register which is used to indicate whether the register currently holds subword data. As shown in Figure 2, when the bit is cleared, the register stores one 32-bit value. When the bit B is set, the two halves of the register store two 16-bit values. The upper half of the register is used to store the speculatively saved value and it can be separately accessed by the four new instructions we add (discussed later). When accessing the value that is definitely present, i.e. when register is accessed by normal instructions, the lower half is accessed.

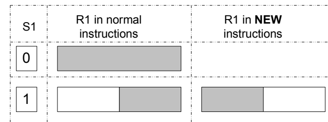

Fig. 2. Accessing registers.

3.2 Instruction Set Support

To speculatively store a value in the upper half of the register and to access it later on, we propose four new instructions. We also add an additional G bit to the status word register which is set and examined by these new instructions. These instructions are described in detail below. The first two instructions are used together to speculatively assign a value to an already occupied register while the last two are used to access a speculatively saved value.

Speculative store: Ssw Rs, addr. This instruction stores the value of `Rs` into the memory address `addr`. In addition, it checks the value for compressibility. In particular, it sets the G bit in the status register if the higher order 16 bits are the same as its 16th bit, i.e. they are sign extensions.

Store move: Smv Rd, Rs. This instruction checks the global condition bit G and the non-speculative value residing in Rd for compressibility. If G is set, and the higher order 16 bits of Rd are the same as its 16th bit, the lower order 16 bits of Rs are moved into upper half of Rd. The B bit of Rd is set to indicate that it now contains two values.

Extract move: Emv Rd, Rs. This instruction checks the B bit of Rs. If the bit is set, the higher order 16 bits of Rs are sign extended and then moved to Rd. In addition, the global condition G is set to indicate that the instruction was successful in finding the speculatively stored value in Rs. If the B bit of Rs is not set, G is cleared to indicate that the value was not found.

Speculative load: Sld Rd, addr. This instruction checks the G bit in the status word. If it is clear, the value stored in memory address `addr` is loaded into Rd; otherwise, the instruction acts as a null instruction. In the latter case the load is not issued to memory because the preceding instruction must have located the speculatively saved value in the specified register.

Let us reconsider the example in Figure 3(a) where value of a is being spilled because no register is free. Let us assume that we would like to speculatively save the spilled value of a in register which holds the value of c (say register $R2$). The code in Figure 3(b) shows how instructions `Ssw` and `Smv` are used to speculatively save the value in $R2$ and then later instructions `Emv` and `Sld` are used to reload the value of a from $R2$ into $R4$. If the speculative save of a is successful, the `Sld` instruction turns into a null operation; otherwise the value of a is reloaded from memory.

(a) original C code

```
/* most of the time
b can be represented
using 16 bits */

a = b + 1;
...

c = a - 10;
```

(d) after SSRA

```
; b → R1_{1..32} or R1_{1..16}
; c → R2_{1..32} or R2_{1..16}
; a → in memory or R2_{17..32}

add R3, R1, 1
Ssw R3, addr_a
Smv R2, R3
...
Emv R4, R2
Sld R4, addr_a
sub R2, R4, 10
```

Fig. 3. Example illustrating the use of new instructions.

From the above discussion it is clear that we pay a cost for speculatively saving and reloading a compressed value. While traditional code would have included a store and a load, in our case we also introduce two extra move instructions (`Smov` and `Emov`). However, the benefits of eliminating loads are much greater as not only loads have longer latency, they can also cause cache misses which leads to even greater delays. Moreover our compiler will only perform speculative register assignment when it is highly likely that it would lead to elimination of loads.

3.3 Hardware Implementation

Other modifications must be made to the processor pipeline when the above mentioned instructions are introduced. Normal instructions must always check the B bit of a register that it reads or writes to. This is needed so that it knows how to interpret and update the contents of the register. Changes are needed when results are being written or operands are being read. For modern embedded processors such as ARM SA-110 (see Figure 4(a)) changes affect the second and fourth stage of the pipeline.

ALU instructions compute the result in *execute* stage while the register is updated at *write back* stage. A small component can thus be added in the intervening *buffer* stage to ensure that the higher order bits are sign extensions. If not, the entire register is used to hold the result and the corresponding B bit of the register is cleared. For memory access instructions that use a register as

the destination, i.e. load instructions, register is ready at the end of *buffer* stage and thus we do not have time to perform validation. We simply clear its B bit and use all the bits in this case.

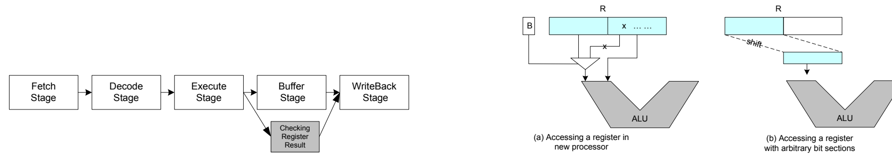

Fig. 4. (a) ARM processor pipeline; and (b) Accessing registers with subword support.

Since the flag bit B of a register determines whether a 16-bit or a 32-bit value is involved in a computation, we need a multiplexer for the higher order 16 bits. Therefore in comparison to a machine without subword support, we introduce extra delay due to the multiplexer shown in Figure 4(b). However, this delay is smaller than the delays in processors that support accesses to arbitrary bit sections such as the Infineon processor [8].

4 Speculative Subword Register Allocation

Given the hardware support designed in the preceding section, we now discuss compiler support needed to carry out speculative subword register allocation (SSRA). Since we have designed our algorithms so that they can be integrated into the `gcc` compiler, we first briefly introduce the implementation of register allocation in `gcc` compiler. Next we describe the details of the three passes that implement SSRA. The *profiling pass* collects information about how likely it is that the two variables can fit into one register and how long is the lifetime during which they coexist. Based on this information, the new *speculative allocation pass* determines a subset of physical registers and speculatively assigns other variables to each of them. The variables picked up in this pass are those that otherwise are spilled into memory. The decision is made to achieve speculatively best performance in terms of reduction in the number of reload operations that are executed. In the *enhanced reload pass*, the compiler generates transformed code utilizing the newly designed instructions.

4.1 Register Allocation in the gcc compiler

The `gcc` compiler [2] performs register allocation in three passes: local register allocation pass, global register allocation pass, and reload pass. These passes operate on the intermediate representation – register transfer language (RTL). Operands are mapped to virtual registers before register allocation. The local and global register allocation passes do not actually modify the RTL representation. Instead, the results of these passes is an assignment of physical registers to virtual registers. It is the responsibility of reload pass to modify the RTL and insert spill code if necessary.

The local register allocation pass allocates physical registers to virtual registers that are both generated and killed within one basic block, i.e. live ranges that are completely local to a basic block are handled in this pass. The allocation is driven by live range priorities. Register coalescing is also performed in

this pass. Since local register allocation works on linear code, it is inexpensive. Local allocation reduces the amount of work that must be performed by the more expensive global allocation pass.

The global register allocation pass allocates physical registers to the remaining virtual registers. This pass may change some of the allocation decisions made during the local register allocation pass. This pass performs allocation by coloring the global interference graph [1]. Virtual registers are considered for coloring in an order determined by weighted counts. If a physical register cannot be found for a virtual register, none is assigned and the virtual register is handled by generation of spill code in the reload pass.

The reload pass replaces the virtual registers references by physical register names in the RTL according to the allocations determined by the previous two passes. Stack slots are assigned to those virtual registers that were not allocated a physical register in the preceding passes. Reload pass also generates spill code for them. Unlike Chaitin-style [1] graph-coloring allocation, which spills a symbolic register, a physical register is spilled. For each point where a virtual register must be in a physical register, it is temporarily copied into a "reload register" which is a temporarily freed physical register. Reload registers are allocated locally for every instruction that needs reloads.

4.2 The SSRA Algorithm

In this section we first discuss how SSRA is integrated into the `gcc` register allocator described above. The details of SSRA are also discussed later. The modified design of `gcc` register allocator after integration of SSRA into it is shown in Figure 5. In integrating SSRA with the `gcc` allocator we keep the following in mind. There are three types of accesses allowed in our architecture: register accesses for values definitely present in registers – these accesses are the fastest; memory accesses that are the slowest; and register accesses for speculatively assigned values which have an intermediate access cost as an additional `Sld` instruction is required.

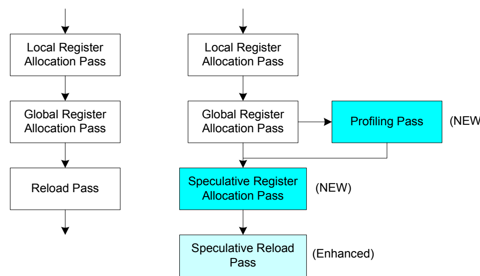

Fig. 5. Framework for speculative register allocation.

In light of the above observation, we first carry out local register allocation and global register allocation passes in exactly the same way. This is because the variables that are referenced more frequently are assigned physical registers in these passes. At run time, the variables that are assigned physical registers in

these phases occupy either the whole physical register or possibly just the lower half. Following the above passes SSRA is used to speculatively assign physical registers to virtual registers that are not colored by the first two passes. Finally, the virtual registers that are still not colored are handled by generating memory references by the reload pass. The reload pass is enhanced to generate the newly designed instructions.

During the speculative allocation pass, the upper halves of the physical registers are made speculatively available for the variables that are not colored by the preceding passes. However, it is important that the savings expected by finding the speculatively assigned values in registers exceeds the additional cost of executing `Emv` instructions when the values are not found in the registers and must be loaded from memory. The savings depend on how frequently the variables are able to be coalesced and how often the references to memory can be avoided. We add a profiling pass which instruments the code and records the information needed to estimate the savings. Since we can only coalesce one variable which already has a register with one variable which does not have a register, we only need to collect profile information for relevant pairs of values. The SSRA pass makes its decisions to achieve speculatively best performance by avoiding maximum number of reload operations. After the decisions are made, a speculative reload pass generates the code according to the decisions made in all of the previous passes. It inserts spill code and the code for speculatively allocated virtual registers. When there are no spilled variables, the behavior of our modified allocator is identical to the behavior of the original `gcc` allocator.

Next we describe the details of the SSRA algorithm. Three main parts of our algorithm are discussed: priority based allocation algorithm; profiling pass details; and the speculative reload pass.

Priority Based Speculative Allocation. The decision to speculatively allocate an occupied register to another variable is made based upon profile information consisting of the following: *coexisting lifetime* of variables $v1$ and $v2$ refers to the period of time during which $v1$ and $v2$ are both live during program execution; and *coalescing probability* of variables $v1$ and $v2$ refers to the percentage of the coexisting lifetime during which $v1$ and $v2$ can be coalesced. The coalesce probability is undefined if two variables never coexist. Note that during program execution one variable may be coalesced with different variables at different program points.

The speculative allocation pass makes use of two interference graphs. One is called the *Annotated Interference Graph* (AIG) which contains the information needed to make coloring decisions while the coloring itself is performed on another interference graph called the *Residual Interference Graph* (RIG). A more detailed description of these graphs follows.

The *Annotated Interference Graph* (AIG) graph is built from the interference graph after global allocation pass in which some nodes are not colored. For each non-colored node, we annotate the edges between this node and its colored neighbors with a 2-tuple (coalescing probability, coexisting lifetime). Figure 6(a) shows a simple example with nine virtual registers and two physical registers.

After global register allocation pass, nodes 1-6 are colored with two colors and nodes 7-9 are not colored. The edge labels are interpreted as follows. Label (0.9, 800) on edge (7,1) indicates that during 90% of 800 units of the time that variables 1 and 7 coexist, both of them are expected to require no more than 16 bits to represent and thus they can simultaneously reside in one register.

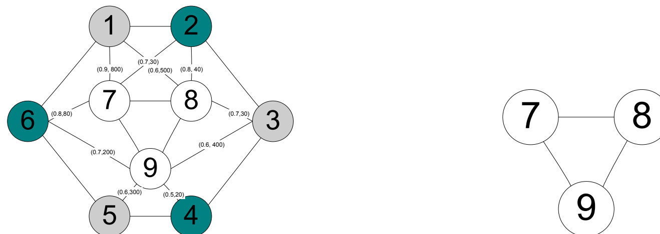

Fig. 6. (a) Annotated interference graph; and (b) Residual interference graph.

The *Residual Interference Graph* (RIG) is a subgraph of AIG that consists of non-colored nodes and edges between them. RIG is not annotated. The RIG for the above example is shown in Figure 6(b). Note that two variables in RIG may be speculatively allocated to the same register if they do not interfere with each other.

The speculative allocation pass colors the nodes in RIG using the annotation information in AIG. After coloring a node in RIG, it shares the same color as at least one of its colored neighbors in AIG. While all colored nodes in AIG are colored in the local or global allocation passes, the colored nodes in RIG are colored in the speculative pass.

Next let us discuss in greater detail how nodes are chosen for coloring from RIG and how colors are selected for them. This process is also priority driven because there are limited physical register resources to which additional variables can be speculatively assigned. Our priority function is based net savings that are expected to result by speculatively assigning a virtual register to a physical register. The savings result from avoiding cost of reloads from memory; however, a cost of one cycle is incurred for each reference due to an extra instruction required when values are speculatively accessed from registers. The savings are also a function of coalescing probabilities. Priority of node n, which is the estimated net savings by speculatively assigning a register to n, is given by:

$$Priority(n) = READ(n) \times READCOST \times MCP(n) - REF(n)$$

where $READ(n)$ is the total number of reads of node n, $REF(n)$ is the total number of references (reads and writes) to node n, $READCOST$ is the number of cycles needed to finish a normal read from memory (i.e., it is the memory latency), and $MCP(n)$ is the maximum coalescing probability of n. Note that the above *Priority* value can be negative for some nodes in which case they are not considered for speculative register assignment. Nodes with higher priority are considered before those with lower priority.

The maximum coalescing probability of n, i.e. $MCP(n)$, is determined by considering all available colors for n and finding which color is expected to result

in most savings. The best choice for a color depends upon the following factors. The higher the coalescing probability of a pair of variables, the more beneficial it is to allocate them to the same register. The longer two variables co-exist, the more beneficial it is to allocate them to the same register. Based upon these two factors we compute the maximum coalescing probability. Moreover we should also note that the physical register being speculatively assigned to a node n in RIG may have been allocated to several non-interfering virtual registers in earlier passes. The coalescing probabilities and lengths of coexistence of each of these virtual registers with n must be considered as long as a virtual register interferes with n. The following equation computes the current maximum coalescing probability $MCP(n)$ for a node n in RIG:

$$MCP(n) = \max_{c \in C(n,RIG)} \frac{\sum_{n' \in Nb(n,AIG) \wedge Cl(n')=c} CLt(n,n') \times CPb(n,n')}{\sum_{n' \in Nb(n,AIG) \wedge Cl(n')=c} CLt(n,n')}$$

Where $C(n, RIG)$ is the set of currently available colors for node n in RIG (i.e., these colors have not been assigned to neighbors of n in RIG), $Nb(n, AIG)$ is the set of neighboring nodes of n in AIG (i.e., these nodes were colored during local or global allocation passes), $CLt(n, n')$ is the length of coexisting lifetime of nodes n and n', $CPb(n, n')$ is coalescing probability of nodes n and n' and $Cl(n')$ is the color assigned to node n'. The max function finds the maximum coalescing probability across all potential colors in $C(n, RIG)$.

Profiling Pass. The speculative allocation pass depends on the coalescing probability and coexisting lifetime profiling information. We implement the profiling by instrumenting the intermediate representation of the code. Profiling is performed after the global allocation pass when the objects of our optimization, those variables which do not get a register, have been identified. At this time the liveness information of the variables at each program points is available since data flow analysis is done before register allocation. The intermediate representation used still contains virtual registers instead of physical register since register reloading pass has not been done.

We are only interested in the relation between colored and non-colored nodes. Whether two variables can be coalesced or not depends on the status of the variables (i.e., whether they fit in 16 bits or not). A variable read will not change the status of the variables and thus consecutive variable reads will share the same coalescing probability. Variable definitions can change the status of variables and thus change the coalescing probability between two variables. Therefore variable definitions play an important role in the coexisting lifetime of two variables. In the priority function described earlier, the number of references has already been considered. Here, we use the length of the status history of two variables to approximate the coexisting lifetime.

Two arrays, $lifetime[i][j][k]$ and $count[i][j][k]$, are used during profiling. Here index i identifies the function, index j identifies the colored variable, and index

k identifies the non-colored variable. The lifetime array records the length of overlap of live ranges of two variables while the count array records the duration over which the two variables are likely coalescable. The coalescing probability is computed by dividing latter by the former.

Speculative Reload Pass. Our speculative reload pass is an enhanced version of the `gcc` reload pass. According to the decisions made in the previous passes, we generate code to access physical registers, access upper halves of physical registers and access memory. In summary, we have three categories of variables to handle in this pass.

First, for variables that are assigned to physical registers in local and global register allocation, the compiler replaces the virtual register names with physical register names in the intermediate representation.

Second, for variables that remain in virtual registers after speculative allocation pass, the compiler allocates slots on the stack and generates spill code. For each definition or use point, we identify a reload register and generate spill code by placing the store after the definition and load before the use.

Finally, for variables that are assigned in the speculative pass, we still need to allocate slots on the stack and generate spill code for them. Instead of using ordinary load and store instructions, we generate speculative load and store instructions. At a definition point, the compiler identifies a reload register and temporarily stores the computed value into this register. A speculative store instruction `Ssw` is generated to speculative store the value back to the stack slot. It is followed by a speculative `Smv` instruction which speculatively moves the value from the temporary register into the upper half of the assigned register.

At each use point, we identify a reload register and try to speculatively extract the value from the assigned upper half of the register using a speculative `Emv` instructions. It is followed by a speculative load instruction `Sld` which actually loads the value from the stack slot if at runtime the required value is not in the register.

5 Experimental Results

We have implemented and evaluated the proposed technique. The speculative subword register allocation algorithms have been incorporated in the `gcc` Compiler for the ARM processor. All the phases have been implemented including the profiling pass, speculative allocation pass, and enhanced register reloading pass to generate spill code. While new instructions are generated, to avoid the effort of modifying the assembler and the linker, we insert `nop` instructions in their place in the code and generate an additional file in which the new instructions and the absolute addresses in the binary where these instructions must replace the `nop` instructions are provided. The code generated by the compiler is executed on the ARM version of the Simplescalar simulator. The simulator was modified to implement the proposed architectural enhancements including the newly incorporated instructions, registers with B bits, the G bit in the status word, and the logic needed to access lower or upper half of the register.

We carried out experiments using some embedded benchmarks from the `Mediabench` and `Commbench` suites. We also took a couple of `SPEC2000` benchmarks to see if the technique we have developed can be useful for general purpose applications. We ran the programs with different memory latencies such that when speculative reloads from registers are successful, the number of *cycles saved* in comparison to reloading from memory is 1, 2, 3 and 4 cycles. Our evaluation is aimed at determining the percentage of dynamic reloads from memory that are successfully transformed into speculative dynamic reloads from registers. We also considered the overall reduction in execution time of the program as a result of avoiding these reloads.

In Table 2 we present the benchmark characteristics. The first six programs are taken from embedded suite while the last two from `SPEC2000`. The table presents the number of residual virtual registers present in the intermediate code generated by the `gcc` compiler following the global register allocation pass, the number of static reloads generated by the `gcc` reload pass, and the number of dynamic reloads that can be attributed to these static reloads during the execution of the benchmarks.

Table 2. Benchmark characteristics.

Benchmarks	Res. Vir. Regs	Static Reloads	Dynamic Reloads
Embedded			
mpeg2.decode	114	483	952060
mpeg2.encode	278	1234	29738119
epic	109	527	6307287
unepic	48	225	107759
g721.encode	9	33	3543596
g721.decode	9	33	4046653
rtr	12	28	3855503
General Purpose			
176.gcc	1755	10200	99248047
164.gzip	77	365	53069259

Table 3. Improvement in cycles.

Saving = 1 Cycle	Saving = 2 Cycles	Saving = 3 Cycles	Saving = 4 Cycles
Embedded			
0.38%	0.65%	0.97%	1.27%
2.53%	4.09%	5.70%	7.33%
0.38%	1.31%	3.06%	5.17%
0.10%	0.21%	0.32%	0.45%
0.95%	1.53%	2.11%	2.71%
1.13%	1.82%	2.53%	3.26%
0.79%	1.20%	1.61%	2.02%
General Purpose			
5.06%	8.97%	13.04%	17.20%
1.08%	1.77%	2.59%	3.42%

The results of studying the effectiveness of our technique in avoiding dynamic reloads are given in Figure 7. We present two numbers for each program: *Speculation percentage* is the percentage of dynamic memory reloads that were changed into speculative dynamic register reloads by our technique; and *Avoidance percentage* is the percentage of dynamic memory reloads for which the speculative register reloads were successful, i.e. the value was found in the register. The above values vary with the savings (1, 2, 3, or 4 cycles) that can be expected from using speculative register reloads. This is because the savings effect the priorities of nodes in *RIG* and therefore the code generated by by the SSRA algorithm. For greater values of savings SSRA will perform speculative register allocation more aggressively.

The results show that on average, the *Speculation percentage* is 57% when the saving is one cycle and 82% when saving is four cycles. Furthermore, when the saving is one cycle, SSRA achieved an average *Avoidance percentage* of 41% (with upper bound of 91% and lower bound of 5%). When the saving is four cycles, *Avoidance percentage* increases to 47% (with upper bound of 93% and

lower bound of 16%). From these results we notice that the minimal savings of one cycle is enough to get most of the reduction in memory reloads.

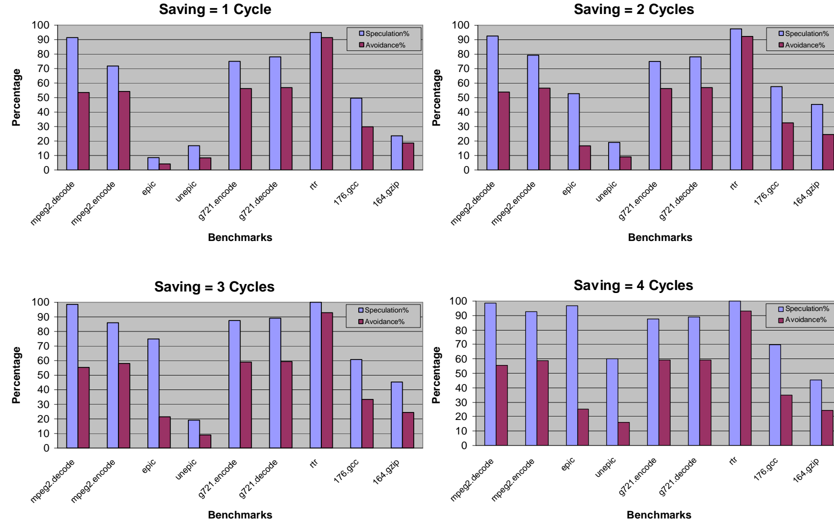

Fig. 7. Effectiveness in avoiding memory reloads.

Table 3 shows the performance improvement in cycles. On average, our method achieved performance improvement of 4.76% when the saving is four cycles and 1.38% when the saving is one cycle. While we had originally designed this technique for embedded applications, we notice that for a general purpose application like `176.gcc` the savings are much higher (5.05% to 17.20%). This is because, due to extensive use of pointers, this benchmark contains a much larger number of dynamic reloads. Moreover narrow width values appear in sufficient abundance in this application for SSRA to be successful.

6 Conclusions

In this paper we presented a technique that exploits presence of narrow width data in programs to more effectively make use of limited register resources in embedded processors. Values otherwise spilled by a coloring allocator are speculatively saved in registers occupied by other variables. Speculative assignment is made such that it is expected that the definitely assigned and speculatively assigned values will be able to simultaneously reside in the same register. We designed a small set of four new instructions through which the feature of speculative register assignment can be implemented without requiring significant amounts of instruction encoding space. The coloring based register allocator was extended by developing a new pass for speculative register allocation. The results of our experiments show that SSRA avoided an average of 47% of dynamic reloads leading to a significant savings in execution time. While our technique was designed for embedded applications, it is also of value for general purpose applications.

References

1. G.J. Chaitin, M.A. Auslander, A.K. Chandra, J. Cocke, M.E. Hopkins, P.W. Markstein, "Register Allocation Via Coloring," *Computer Languages*, 6(1):47-57, 1981.
2. C.E. Foster and H.C. Grossman, "An Empirical Investigation of the Haifa Register Allocation in the GNU C Compiler," *IEEE Southeast Conference*, pages 776-779, vol.2 , April 1992.
3. R. Gupta, E. Mehofer, and Y. Zhang, "A Representation for Bit Section based Analysis and Optimization," *International Conference on Compiler Construction* (CC), pages 62-77, Grenoble, France, Apr 2002.
4. C. Lee, M. Potkonjak, and W.H. Mangione-Smith, "Mediabench: A Tool for Evaluating and Synthesizing Multimedia and Communication Systems," *IEEE/ACM International Symposium on Microarchitecture* (MICRO), December 1997.
5. B. Li and R. Gupta, "Simple Offset Assignment in Presence of Subword Data," *International Conference on Compilers, Architecture, and Synthesis of Embedded Systems* (CASES), San Jose, CA, October 2003.
6. 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), pages 69-78, Grenoble, France, October 2002.
7. J. Lin, T. Chen, W.C. Hsu, and P.C. Yew, "Speculative Register Promotion Using Advanced Load Address Table (ALAT)," *International Symposium on Code Generation and Optimization* (CGO), 2003.
8. 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.
9. 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, 2000.
10. D. Seal, Editor, "ARM Architectural Reference Manual," Second Edition, Addison-Wesley.
11. S. Tallam and R. Gupta, "Bitwidth Aware Global Register Allocation," *30th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL), pages 85-96, New Orleans, LA, January 2003.
12. 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.
13. T. Wolf and M. Franklin, "Commbench - A Telecommunications Benchmark for Network Processor," *IEEE International Symposium on Performance Analysis of Systems and Software* (ISPASS), April 2000.
14. J. Yang and R. Gupta, "Energy Efficient Frequent Value Data Cache Design," *IEEE/ACM 35th International Symposium on Microarchitecture* (MICRO), pages 197-207, Istanbul, Turkey, November 2002.
15. Y. Zhang and R. Gupta, "Data Compression Transformations for Dynamically Allocated Data Structures," *International Conference on Compiler Construction* (CC), pages 14-28, Grenoble, France, Apr 2002.
16. Y. Zhang and R. Gupta, "Enabling Partial Cache Line Prefetching Through Data Compression," *International Conference on Parallel Processing* (ICPP), Kaohsiung, Taiwan, October 2003.