Allocating Architected Registers Through Differential Encoding

XIAOTONG ZHUANG and SANTOSH PANDE
Georgia Institute of Technology

Micro-architecture designers are very cautious about expanding the number of architected and exposed registers in the instruction set because increasing the register field adds to the code size, raises the I-cache and memory pressure, and may complicate the processor pipeline. Especially for low-end processors, encoding space could be extremely limited due to area and power considerations. On the other hand, the number of architected registers exposed to the compiler could directly affect the effectiveness of compiler analysis and optimization. For high-performance computers, register pressure can be higher than the available registers in some regions. This could be due to optimizations like aggressive function inlining, software pipelining, etc. The compiler cannot effectively perform compilation and optimization if only a small number of registers are exposed through the ISA. Therefore, it is crucial that more architected registers are available at the compiler's disposal, without expanding the code size significantly.

In this article, we devise a new register encoding scheme, called differential encoding, that allows more registers to be addressed in the operand field of instructions than the direct encoding currently being used. We show that this can be implemented with very low overhead. Based upon differential encoding, we apply it in several ways such that the extra architected registers can benefit the performance. Three schemes are devised to integrate differential encoding with register allocation. We demonstrate that differential register allocation is helpful in improving the performance of both high-end and low-end processors. Moreover, we can combine it with software pipelining to provide more registers and reduce spills.

Our results show that differential encoding significantly reduces the number of spills and speeds-up program execution. For a low-end configuration, we achieve over 14% speedup while keeping code size almost unaffected. For a high-end VLIW in-order machine, it can significantly speed-up loops with high register pressure (about 80% speedup) and the overall speedup is about 15%. Moreover, our scheme can be applied in an adaptive manner, making its overhead much smaller.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Code generation; compilers; optimization*; G.2.2 [**Discrete Mathematics**]: Graph Theory

General Terms: Languages, Design, Algorithms, Performance

Additional Key Words and Phrases: Register allocation, architected register, differential encoding

A preliminary version of this article will appear in *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI)* [Zhuang and Pande 2005].
Authors' addresses: X. Zhuang and S. Pande, College of Computing, Georgia Institute of Technology, 801 Atlantic Drive, Atlanta, GA 30332; email: xt2000@cc.gatech.edu.

ACM Reference Format:
Zhuang, X. and Pande, S. 2007. Allocating architected registers through differential encoding. ACM Trans. Program. Lang. Syst. 29, 2, Article 9 (April 2007), 46 pages. DOI = 10.1145/1216374.1216377 http://doi.acm.org/10.1145/1216374.1216377.

1. INTRODUCTION

The width of the register field in an instruction determines the number of architected registers that can be managed by the compiler. In other words, the compiler can only allocate registers that are exposed to it through the ISA (instruction set architecture), although physical registers are normally more than architected ones. Micro-architecture designers are very cautious about expanding the number of architected registers (or the register field in the ISA) due to a number of reasons.

First, the register field directly affects the code size. Notice that adding one bit to the register field typically leads to an increase of two or more bits for each instruction due to multiple register fields. We observe that register fields take a significant portion of the code size, since most instructions in the generated code use register(s) one way or another. For example, register field takes about 28% of the Alpha binary and 25% of the ARM binary. Therefore, the size of the register field heavily influences the size of generated code. Second, the memory footprint of a program also affects the memory traffic to the code segment and determines the access pressure on the I-cache. If instructions become wider, higher pressure is imposed on the I-cache. Moreover, wider instructions also complicate the decode stage in the pipeline, stretching clock cycles, increasing die size and power consumption, etc.

This is especially evident for low-end processors, where encoding space could be very limited due to tight area and power constraints. Typically, instruction cache accounts for a big portion of the die area and overall power consumption. A recent power consumption analysis on the ARM processor reveals that cache power accounts for about half of the power budget on ARM. Moreover, the I-cache (with the same size as D-cache) consumes about 40% more power than the D-cache [Segars 2001]. Thus, the number of architected registers is limited to avoid instruction-width growth. Even if the hardware can support more registers, the number of architected registers defined by ISA is much smaller due to these encoding issues. For example, the Motorola DSP 56300 chip [Motorola 2000] only has six registers available for ALU instructions and the register numbers are compactly encoded. StarCore SC 110 contains sixteen registers but only exposes either the HI (top eight) or LO (lower eight) registers at-a-time to a given instruction. Intel's StrongARM processor provides two sets of ISAs: 32-bit ARM ISA and 16-bit THUMB ISA [Intel 1998; ARM 2007]. Most THUMB instructions can only access eight registers although all sixteen registers are physically present. The compact THUMB instruction set results in better I-cache power savings, but lower performance, partially due to the small number of architected registers available [Krishnaswamy and Gupta 2002].[1] Study in Krishnaswamy and Gupta [2002] shows that compiling the

[1]Although a few instructions, like MOV, can access all 16 registers, the majority of THUMB

same program with a 16-bit THUMB mode instead of the 32-bit ARM mode can save up to 19% I-cache energy. Similar to ARM/THUMB, the MIPS-16 [MIPS 2001] processor also adopts two instruction sets of different widths.

However, restricting the number of architected registers typically incurs performance penalty, since compiler-managed register allocation can only utilize the exposed architected registers. Limiting architected registers impedes compiler optimizations. This is especially true for VLIW machines, where the compiler plays a crucial role in optimizing the code. Typically for superscalar machines, the number of architected registers does not affect register allocation to scalar variables much, since in most cases register pressure is lower than the number of architected registers. However, in certain circumstances, register pressure could be very high. For example, if functions are aggressively inlined to reduce function call overhead, high register pressure regions may appear. Software pipelining is performed to pipeline loops, which might increase register pressure as well. Therefore, a flexible and low-overhead register encoding scheme is desirable to effectively increase the number of architected registers for such code segments, while being easily turned off for the rest of the code with low register pressure.

As discussed earlier, for low-end processors, the number of spills is very sensitive to the number of architected registers. More spills lead to more accesses to the memory subsystem, incurring longer delays and power consumption in the D-cache. Although the hardware can support more registers such as on ARM and MIPS, the ISA bottleneck prevents them from being addressed and utilized. Krishnaswamy and Gupta [2002] mention that with a compact ISA (THUMB or MIPS-16), instruction count can increase by 9% to 41%, which not only slows down the execution but eliminates some of the benefits brought about by a compact ISA.

Due to the aforementioned reasons, we find it necessary to devise a performance-driven approach to work around the ISA bottleneck due to architected registers. This is particularly important for processors with tight encoding space. In this article, we first propose a new encoding scheme called differential encoding. Instead of encoding register numbers directly, differential encoding encodes the difference between register numbers of the current and previous accesses. Differential encoding can potentially address more registers than does direct encoding. We first show that the hardware cost to implement such encoding is very low. We then present several approaches that make use of the extra architected registers, including performing better register allocation, reducing spills for software pipelining, etc. Our results show that differential encoding significantly reduces the number of spills and speeds-up program execution for a number of benchmarks. Especially for a low-end configuration, we can achieve over 14% speedup while keeping code size almost unaffected. This also significantly accelerates loops with high register pressure (about 80% speedup) on a high-performance machine model.

The rest of the article is organized as follows: Section 2 introduces the differential encoding scheme; Section 3 provides an overview of our approach;

instructions only see 8 registers.

Section 4 discusses how to model this problem and our optimization objective; Sections 5, 6, and 7 give the details of the three approaches and how they are combined with register allocation; Section 8 proposes ways to make differential register allocation more adaptive; Section 9 suggests two auxiliary optimization techniques; Section 10 talks about its application to software pipelining and Section 11 addresses a number of other considerations; Section 12 evaluates the algorithms; Section 13 discusses the related work; and Section 14 concludes the work.

2. DIFFERENTIAL REGISTER ENCODING

To simplify the discussion and illustration, we assume a RISC ISA and that the registers under consideration are uniform and classless. In Section 11, we will discuss the case of registers forming multiple classes.

Traditionally, register numbers are encoded directly in the register field, that is, to encode R_5, we just put 5 in the register field. By contrast, differential register encoding encodes the difference between the currently accessed register number and the previously accessed one. Therefore, with differential register encoding, the register field encodes "differences". For example, assuming that we want to access registers R_1, R_3, and R_8 in that order, the encoded differences are then 2 (from R_1 to R_3) and 5 (from R_3 to R_8). Notice that encoding differences does not save any encoding space because the number of differences is not smaller than that of the original register numbers. In this work, we first force differences to take a smaller range (i.e., less numbers of differences are allowed); then we handle cases where encoding is not possible with special instructions.

To encode *RegN* number of registers, the register field should have at least $RegW = \lceil \log_2 RegN \rceil$ bits. We call this most widely-used scheme *direct encoding*. By contrast, suppose that we encode *DiffN* distinct differences (for ease of hardware implementation, we restrict differences to [0, *DiffN*-1]). It will then take $DiffW = \lceil \log_2 DiffN \rceil$ bits. Our goal is to make $DiffW < RegW$ while retaining the property that all *RegN* registers can be properly addressed with little cost.

First of all, differential register encoding assumes registers are accessed in a certain sequence such that we can distinguish between "current access" and "previous access". Registers are decoded according to the scheme before they are actually accessed. Therefore, we must define a nominal *register access order* or *access order*. Naturally, registers should be accessed by following the instruction order. For register fields in the same instruction, they should follow a certain predetermined order, for example, access source operands one-by-one then the destination operand: src1, src20., dst. Access order must be agreed upon beforehand to make the encoding and decoding work consistently. Based on the access order, we can construct the *register access sequence* or *access sequence* from the original code.

Formally, suppose the code has register access sequence as follows: $R_{n_1}, R_{n_2} \ldots R_{n_k}$, where $n_1, n_2 \ldots n_k$ are register numbers (which will be put in the register fields under direct encoding). Also, assume $n_0 = 0$. Under

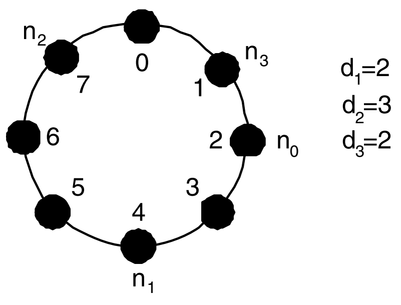


Fig. 1. Calculate the difference.

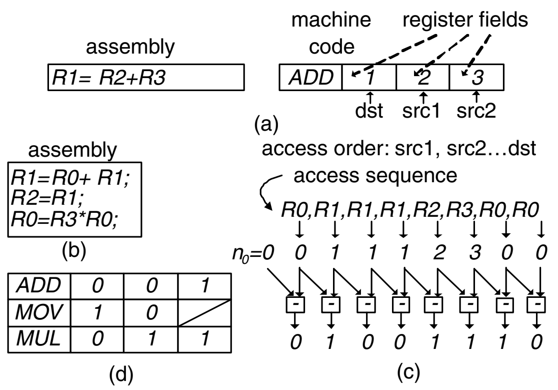


Fig. 2. Illustrations for direct encoding and differential encoding.

differential encoding, the encoded numbers are $d_1, d_2 \dots d_k$ such that

$$d_i = (n_i - n_{i-1}) \bmod RegN. \tag{1}$$

To decode, we should do

$$n_i = (d_i + n_{i-1}) \bmod RegN. \tag{2}$$

For clarity, we define the modulo (mod) operation as follows.

Definition 1 (*Modulo Operation*). If $A \bmod M = B$, where M is a positive integer, then there is an integer k such that $A - B = kM$ and $B \in [0, M - 1]$. Examples are: $4 \bmod 3 = 1$, $-1 \bmod 3 = 2$.

Therefore Eq. (1) gives a value from 0 to $RegN-1$. In other words, if $n_i \geq n_{i-1}$, then $d_i = n_i - n_{i-1}$, otherwise $d_i = n_i - n_{i-1} + RegN$. An illustration is shown in Figure 1. If we order $0, 1 \dots RegN - 1$ on a clockwise circle, d_i is actually the number of hops from n_{i-1} to n_i following the clockwise direction. Here n_1 is two hops away from n_0, n_2 is three hops away from n_1, and n_3 is two hops away from n_2.

In Figure 2(a), direct encoding simply puts the register numbers into the corresponding register fields, for example, if the register is R_1, we simply put 1. Figures 2(b) and (c) illustrate differential encoding. First, we define the register access order as src1, src2 ... dst. Thus the assembly code in Figure 2(b) yields the access sequence as shown in Figure 2(c) and the final encoding as in Figure 2(d). The bottom of the figure shows the encoded differences. As shown in this example, under direct encoding we need to encode register numbers from zero to three, that is, $RegN = 4$, and two bits should be allocated for each register field, that is, $RegW = 2$. Notice that with differential encoding, we only need to encode two different numbers: zero and one. In other words, the register field only needs one bit, namely, $DiffN = 2$, $DiffW = 1$. However, all four registers can be properly addressed as in the original code. This achieves 50% reduction of encoding space in the address field.

In short, differential encoding uses *DiffW* bits but can address *RegN* registers, where $DiffW < RegW = \lceil \log_2 RegN \rceil$, so we are able to save encoding space. Alternatively, with a given amount of encoding space we are able to access more registers than with direct addressing. Notice that as long as instructions are decoded in order (which is almost always true), differential encoding is applicable.

2.1 Implementation Overhead

Before discussing compiler techniques that can exploit differential encoding, we first show that its implementation overhead in hardware is very low.

To decode an instruction generated with differential register encoding, we need a storage space to store the register number that is accessed the previous time. We call this the *last_reg* register, namely, the n_{i-1} in Eq. (2), which should contain *RegW* bits. To decode an instruction, we decode the register fields one-by-one according to the access order (modulo), add the difference stored in the register field to *last_reg*, get the decoded register number, and then assign it back to *last_reg* in order to decode the next register field. The overhead due to this register is negligible even for embedded processors. For superscalar machines, multiple *last_reg* registers are needed if the processor speculatively fetches and executes on several paths. The cost grows linearly with the number of concurrent execution paths because differential decoding can be performed independently on those paths. Given that speculating on multiple paths already demands a lot more resources (the cost actually grows superlinearly), the additional overhead due to this scheme is less of a concern.

Eq. (2) tells us that the basic hardware component is the *modulo adders*. Each modulo addition can be split into one addition and one modulo operation. Modulo operation is complicated in general, however, in our case it can be significantly simplified. If *RegN* is a power of two, the modulo operation simply truncates the value to its lower *RegW* bits. Otherwise, there are two cases according to Eq. (2). Notice that $d_i \in [0, DiffN - 1]$ and n_{i-1}(i.e., *last_reg*) $\in [0, RegN - 1]$. Since $RegN \geq DiffN$, the sum of these two values must be in the range $[0, 2 \times RegN - 1]$. Therefore, if the sum is less than *RegN*, nothing else needs to be done, otherwise we deduct *RegN* from it so the final result is in $[0, RegN - 1]$.

Eq. (2) shows that we must decode register fields sequentially, that is, only after one register field is decoded can we decode the next. Since decoding is on the critical path, we should speed it up as much as possible. Actually, by modifying the formula slightly, *all operands can be decoded in parallel*. Supposing there are two operands in one instruction, encoded as d_1 and d_2, we can decode them in parallel as follows:

$$\begin{aligned} n_1 &= (last\_reg + d_1) \bmod RegN \\ n_2 &= (last\_reg + d_1 + d_2) \bmod RegN. \end{aligned}$$

Compared with sequential decoding, we need more modulo adders. Therefore, it becomes extremely important that the modulo adder is implemented efficiently. Note that modulo adders are very similar to normal adders. For differential decoding, values involved in modulo addition are typically small. For embedded processors with 16 registers, the adder only needs to handle 4-bit input/outputs which can be simply implemented with two-level combinational logic. Such circuits only incur two-gate delay. According to HSPICE, it is less than 0.4 ns, that is, 1/5 cycle if the processor is clocked at 500MHz. For two input adders, we need to build an 8-bit input and 4-bit output combinational circuit. For three input adders, a 12-bit input and 4-bit output combinational circuit is required. Even for the latter, a rough estimation tells us that it can be built with less than 2k transistors, which is negligibly small.

For superscalar machines, the number of architected registers increases a lot, for example, Alpha 21264 has 32 integer registers and Itanium has 128 registers. Also, multiple instructions might be decoded in each cycle. However, even with 128 registers, 7-bit modulo adders can be constructed easily, given that typical adders on those machines are 64-bit. As a matter of fact, Intel's adders can now be clocked at 3–4 GHz. In addition, operand decoding is normally conducted in parallel with operator decoding, therefore its latency can be mostly hidden. Thus the delay due to differential decoding should not be a concern at all. Finally, differential encoding only adds slight complexity to the decode stage. It is entirely transparent to the rest of the processor pipeline.

2.2 Two Important Issues

The preceding discussion may lead to the perception that differential register encoding is always beneficial. Actually, differential register encoding is not always applicable in a straightforward manner. In this subsection, we discuss two important issues: "difference out of range" and "multipath inconsistency".

2.2.1 *Difference out of Range.* Sometimes the difference between register numbers can go out of range. For instance, in the previous example of Figure 2(b), if the first instruction is $R_1 = R_0 + R_2$, we need to encode $(2 - 0) \bmod 4 = 2$ for the second source operand, which means we must allow encoding the difference of 0, 1, and 2. If we allow d_i to take three different values, the register field must be extended to two bits, that is, $DiffW = 2$. This will completely eliminate the benefits of differential encoding since $DiffW = RegW$. In practice, we get all d_i values from Eq. (1) to see if $DiffW$ can be smaller than $RegW$. Naturally, we hope that $d_i \in [0, DiffN - 1]$ instead of $[0, RegN - 1]$. In case some d_i's are out of this range, namely, $d_i \in [DiffN, RegN - 1]$, we can still enforce $DiffW < RegW$ and handle out-of-range d_i's afterwards, that is, we use special instructions to tackle out-of-range cases separately—details will be given in Section 2.3.

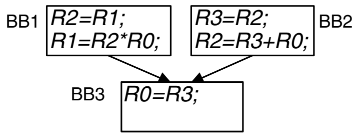


Fig. 3. Example for multipath inconsistency.

2.2.2 *Multipath Inconsistency.* During decoding, we always need a unique n_{i-1}, that is, the previous register number in the *last_reg*, to decode the current register number. This is quite straightforward for sequential code since n_{i-1} is always uniquely determined. However, this may be an issue at control flow join points. Multiple paths from different predecessors of a basic block may have different n_{i-1}'s stored in *last_reg*. This causes inconsistency for decoding. As shown in Figure 3, if we assume the access order to be src1, src20.dst, there could be two possible values in *last_reg* at the entry point of BB3. If the execution goes from BB1 to BB3, then *last_reg* $= 1$, since R_1 is the last register in the access sequence at this point. If the execution goes from BB2 to BB3, then *last_reg* should be 2. Remember that register fields contain differences instead of absolute register numbers, therefore we must rely on a unique *last_reg* to decode properly. Multiple execution paths to the same instruction with different values in *last_reg* could be problematic.

2.3 ISA Extension

To solve both problems mentioned before, a special instruction is introduced to change the value in *last_reg*. We call it *set_last_reg*. Two parameters can be passed to *set_last_reg*. The complete format is as follows:

set_last_reg(value) *set_last_reg(value, delay_num)*

The first type of *set_last_reg* just assigns the value to *last_reg*, while the second type sets the value after a certain number of *register fields* are decoded. The second type is particularly useful when we want to change *last_reg* in the middle of decoding an instruction. For example, instruction $R1 = R0 + R2$ cannot be differentially encoded because the difference between the first and second source operands is larger than 1 (assume *DiffN* $= 2$). We can put *set_last_reg*$(2, 1)$ in front of this instruction. In this manner, after encoding source operand 1, *last_reg* is set to 2, therefore the second field can be encoded as 0, which is less than *DiffN*.

The mutipath inconsistency problem can be solved as well. For the example in Figure 3, we can insert a *set_last_reg* at the entry point of BB3, so that *last_reg* will contain the same value regardless of where the control flow comes from. Alternatively, we can insert such instruction at the end of one or more predecessors to maintain a consistent *last_reg* value when the execution reaches the join point.

The real implementation of *set_last_reg* can be simplified in several aspects to reduce overhead. First, the range of *delay_num* is restricted such that the decoder does not need to keep track of each *set_last_reg* for a long time, since the purpose of *delay_num* is to alter *last_reg* in the middle of an instruction and the number of register operands in an instruction is typically small. Thus a few bits are enough for *delay_num*. Secondly, we can utilize the register field to save the bits for *value* as well. Intuitively, the width of *value* should be *RegW*, therefore it cannot be fit into the register field which is *DiffW* bit wide. However, with the *set_last_reg* instruction, the next register field is actually wasted (i.e., number zero is always put in it). Going back to the previous example of $R1 = R0 + R2$, once we set *last_reg* to 2, the register field originally used for encoding $R2$ is put a trivial, number, namely, 0. This observation actually holds for both "difference out of range" and "multipath inconsistency" where *set_last_reg* instructions are needed. A simple improvement is to put some bits of *value* to the register field. We found this is extremely effective, since normally *RegW* is only a few bits more than *DiffW* because the *RegN*/*DiffN* ratio shouldn't be too big (as indicated in our experiements, a big ratio incurs large number of extra instructions; we always pick *RegN*/*DiffN*$2 < 2$, that is, *value* is actually one bit wider). Thus, both *value* and *delay_num* are narrow. As a matter of fact, we can often pack two or more *set_last_reg* instructions into one instruction to save encoding space.

During program execution, all *set_last_reg* instructions only appear until the decode stage. Once register fields are decoded properly, *set_last_reg* instructions are removed from entering the execution stage. In other words, for all pipeline stages after decoding, they do not exist. The *set_last_reg* instruction is as in expensive as a move instruction. It actually moves an immediate value to *last_reg*. In the decode stage, if there is no *delay_num*, the *set_last_reg* instruction is performed immediately to change the *last_reg*, otherwise the *delay_num* and value are stored in a FIFO queue until the particular register field is to be decoded. The size of the queue depends on the width of *delay_num* and the width of *value*. As discussed earlier, *value* is normally one bit; if *delay_num* is from 1 to 8, we need $8 \times (3 + 1) = 32$ bits for the entire queue, which is the size of one register for any 32-bit system.

3. OVERVIEW OF OUR APPROACH

So far, we have shown that differential encoding can expose more architected registers to the compiler. The previous section hints at how to convert a program after register allocation to the one that uses differential encoding. However, simply encoding the program with differential encoding could introduce many *set_last_reg* instructions, as indicated in the evaluation section. Therefore, it is important that the increased architected registers can be fruitfully utilized toward performance improvements by overcoming the extra cost due to *set_last_reg* instructions.

We will show several optimization techniques and their applications. First, we look at how to enhance the register allocation in general through three approaches: (1) remap the register numbers for differential encoding after register allocation is done; (2) modify the select stage of a graph-coloring-based register allocator; and (3) combine it with the coalescence and select stages of an optimal spilling register allocator [Appel and George 2001].

Notice that the optimization's impact increases from the first approach to the last. Also, the latter ones can make use of the former, that is, they can be

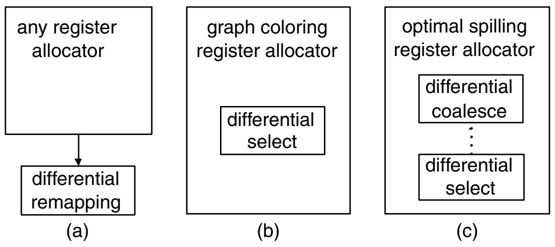


Fig. 4. Overview of the three approaches.

combined. Figure 4 illustrates the three approaches. Figure 4(a) shows the most simple approach, called *differential remapping*. Differential remapping can follow any register allocator, therefore it is a postpass approach. It remaps register numbers that are given by the register allocator and attempts to maximize the benefits of differential encoding. Figure 4(b) shows the second approach, called *differential select*. We assume a graph-coloring-based register allocator such as Chaitin et al. [1981], Briggs et al. [1994], and George and Appel [1996]. The algorithm is built into the select stage to select register numbers that are beneficial to differential encoding.

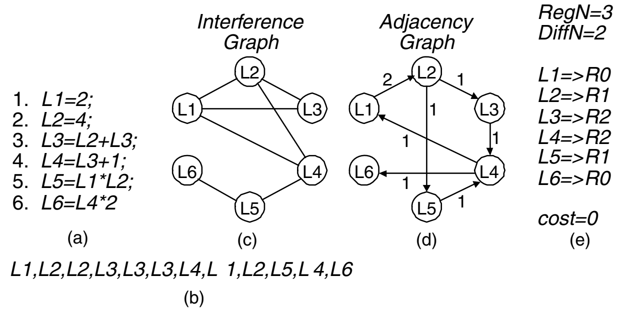
Finally, the third approach is based on an optimal spilling register allocator [Appel and George 2001]. After the optimal number of spills are decided, we construct an interference graph which is provably colorable with the number of registers available (therefore no extra spills get generated) but we still need to remove a large number of move instructions through coalescence. The original algorithm does aggressive coalescence and graph coloring. If the coalescence makes the graph uncolorable, they undo it. We propose differential coalesce which is combined with the coalesce stage to favor differential encoding. In other words, we need to reduce the number of both moves and *set_last_reg* instructions. Our approach attempts to merge these two goals. Also, differential select is invoked during the select stage. Furthermore, differential remapping can always be invoked after approach (2) or (3), since the two have been integrated into register allocation, whereas differential remapping is a postpass optimization.

In Section 8, we will propose an adaptive scheme which changes *RegN* at different program points. Finally, Sections 9 and 10 give several auxiliary optimizations and applications for differential register allocation.

4. PROBLEM MODELING

To integrate differential encoding into register allocation, we need to build not only the interference graph, but an *adjacency graph*. The adjacency graph captures the adjacency of (register-resident) variable accesses. We define adjacency graph as follows.

Definition 2 (*Adjacency Graph*). An adjacency graph is a directed weighted graph, each node representing a live range or a register. An edge from node v_i to v_j with weight w_{ij} means live range or register v_j has followed v_i immediately for w_{ij} times in the access sequence.



```
1. L1=2;
2. L2=4;
3. L3=L2+L3;
4. L4=L3+1;
5. L5=L1*L2;
6. L6=L4*2
```
(a)

L1,L2,L2,L3,L3,L3,L4,L 1,L2,L5,L 4,L6

(b)

RegN=3
DiffN=2

L1=>R0
L2=>R1
L3=>R2
L4=>R2
L5=>R1
L6=>R0

cost=0

(e)

Fig. 5. Example for problem modeling.

Notice that the adjacency graph can be built during or after register allocation (as used in differential remapping). If built during register allocation, the nodes are live ranges.[2] If it is built after register allocation, each node represents a register that has been allocated to one or more live ranges. Therefore the adjacency graph can be more restrictive for postpass optimizations, since multiple live ranges might be assigned to the same register, leading to more edges being linked to one node.

As an example, Figure 5(a) shows a piece of code with six live ranges $L1$ to $L6$. The access sequence is shown in Figure 5(b), given an access order, an of src1, src2 … dst. We assume only $L5$ and $L6$ are live after the end of the code segment and the interference graph is shown in Figure 5(c). Based on the access sequence and our definition of the adjacency graph, we can build the adjacency graph in Figure 5(d).

We observe that there is edge with weight 2 between $L1$ and $L2$ because $L1$ and $L2$ appear consecutively in the access sequence twice. Similarly, "$L2, L3$", "$L3, L4$", "$L4, L1$", "$L2, L5$", "$L5, L4$", "$L4, L6$" all appear once in the access sequence, which are shown as edges with weight 1 on the adjacency graph. Also notice that we do not draw any self-looped edge on the same node for adjacent access pairs like "$L2, L2$", "$L3, L3$" because they are always covered with differential encoding, that is, we can specify the difference to be 0 and put 0 in the register field.

Next, we want to address the objective function associated with the differential register allocation. Eventually each node will be assigned a register number. Traditional register allocation only ensures that the same register should not be assigned to interfering ranges. However, with differential encoding, we are not only interested in enforcing the aforementioned restriction, but are also concerned with the actual register numbers that are assigned to live ranges. Intuitively, the adjacency graph specifies the requirements for the register numbers that are assigned to two adjacently accessed nodes. Suppose there is an edge from node v_i to node v_j with weight w_{ij}. If the register number for v_i is $reg\_no(v_i)$ and the register number for v_j is $reg\_no(v_j)$, they should satisfy

$$0 \leq (reg\_no(v_j) - reg\_no(v_i)) \bmod RegN < DiffN. \tag{3}$$

[2]Sometimes these are called virtual registers.

Only when condition (3) is satisfied can we use differential encoding to encode v_i when the previous access is to v_j. Otherwise, we need extra *set_last_reg* instructions to adjust *last_reg* after accessing v_i. Also we say the edge from node v_i to node v_j is *covered* by this register assignment. The weight of the edge represents the number of times v_j immediately follows v_i. Thus, if condition (3) cannot be satisfied, we will need w_{ij} *set_last_reg* instructions. Thus as an objective function, we must minimize the *total cost* which is the *sum of edge weights in adjacency graph that do not satisfy condition* (3). For the example in Figure 5, we give an optimal solution in Figure 5(e), assuming $RegN = 3$ and $DiffN = 2$. Notice that all edges on the adjacency graph satisfy condition (3), therefore we need no extra instructions for differential encoding. Formally, the optimization objective is expressed as follows.

OPTIMIZATION OBJECTIVE *Devise a register allocation scheme to all live ranges which not only considers the objective of a traditional register allocation, but also tries to minimize the sum of edge weights on the adjacency graph that do not satisfy condition* (3).

Notice that differential register allocation is built on top of a traditional register allocator, which means it must first satisfy the requirements of a register allocation, algorithm such as: at any program points, colive live ranges should not be assigned with the same register and the number of colive live ranges should not exceed the total number of registers. Since the number of *set_last_reg* instructions is also an extra cost involved in register allocation, we consider this together with other costs such as the number of spills.

Many complicated register allocation schemes have been proposed in literature. Techniques such as variable coalescence [Chaitin et al. 1981] can change the shape of both interference graphs, and adjacency graphs, however, we can always modify the adjacency graph together with the interference graph and get the cost for differential register encoding. Therefore, the adjacency graph is a viable means of help with a traditional register allocator for the purpose of differential encoding.

The example in Figure 5 is only a piece of sequential code. For adjacent access pairs that cross the basic block boundary, namely, from the last access in the predecessor to the first access in the current basic block, we can set the edge weight differently. Although there could be multiple edges from the last accesses of several predecessors of a basic block, actually we need (at most) one *set_last_reg* instruction at the beginning of the basic block. Thus, in such cases we divide the add-on weight of the edge by the number of predecessors of the basic block.

Finally, profile information could be incorporated to improve the cost estimation. Different adjacent access pairs have different execution frequencies. For a better estimation, the frequency should be reflected in the edge weights. However, profile information is not always available and varies across different runs. Thus, our goal was to keep compiler methods that work on as simple and consistent a cost model as possible to avoid this skew.

5. DIFFERENTIAL REMAPPING

Our first approach attempts to remap the register numbers after a register allocator has completed register allocation. As mentioned in Section 2, we temporarily assume that the registers under consideration are uniform and classless. Differential remapping permutes the register numbers and thus does not change the solution given by a traditional register allocator (which only enforces the constraint that colive ranges should get different registers). However, permuting the register numbers could lead to different costs for differential register encoding and this is what we attempt to optimize in the postpass.

A simple example is shown in Figure 6. The interference graph is in Figure 6(a); we have 3! ways to assign registers to the live ranges– (Figure 6(c)). In other words, as long as the three live ranges are assigned to different registers, the solution is acceptable for a traditional register allocator. Using the adjacency graph shown in Figure 6(b), we calculate the cost as described in the last section (according to condition (3)), assuming *RegN* = 3 and *DiffN* = 2. The costs under different permutations are different, since the actual register numbers do matter for differential encoding. We summarize the preceding discussion as the following *Permutation Equivalence lemma*.

LEMMA (PERMUTATION EQUIVALENCE). *If the (architected) registers under consideration are uniform and registers are allocated as follows:* $L1 \rightarrow R_{n_1}, L2 \rightarrow R_{n_2} \ldots$ *with a traditional register allocator, then for any permutation* P *of all the architected registers, we will have* $L1 \rightarrow R_{P(n_1)}, L2 \rightarrow R_{P(n_2)} \ldots$ *as another legal solution of register allocation. Depending on the permutation, the cost on the adjacency graph may vary.*

We omit the proof of this lemma, since it is quite straightforward. The lemma says that no permutation can affect the interference graph due to the fact that if two live ranges are assigned to different registers before the permutation, they will get different registers afterwards. Therefore permuting registers only affects the adjacency graph. As illustrated in Figures 6(c) and (d), after permutation, although the register numbers assigned to the nodes of the adjacency graph have changed, it still meets the requirement of the interference graph, therefore the solution is still valid but costs are different on the adjacency graph. There are three registers, thus we will have 3! solutions, as shown in Figure 6(c). Notice that when some live ranges are precolored, for example, due to caller/callee conventions, the lemma needs some changes. We will address this in Section 11.

Also, we notice that there are actually two groups of equivalent solutions in Figure 6(c). One group gets cost zero and the other's cost is two. Each group consists of three permutations. This prompts the following *Modulo Equivalence lemma*.

LEMMA (MODULO EQUIVALENCE). *If the registers under consideration are uniform and one (differential) register allocation solution is* $L1 \rightarrow R_{n_1}, L2 \rightarrow R_{n_2}, \ldots,$ *then we have another solution with the same cost:* $L1 \rightarrow R_{(n_1+k) \bmod RegN}, L2 \rightarrow R_{(n_2+k) \bmod RegN} \ldots,$ *where* k *is an integer constant.*

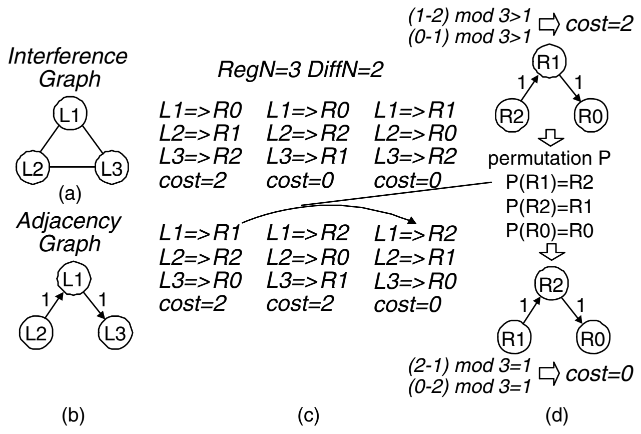


Fig. 6. Example for register remapping.

The modulo Equivalence lemma says that we can shift all register numbers by a constant value, take modulo with respect to *RegN*, and get an equivalent solution with equal cost on the adjacency graph. Obviously, each solution has *RegN* − 1 solutions that are modulo equivalent to it. The *RegN* solutions form a solution group. Thus, in Figure 6(c), we have two solution groups, each with three solutions.

Differential remapping comes after register allocation, therefore all live ranges have been assigned register numbers. It attempts permutations of register numbers, as illustrated in Figure 6. First of all, we take the code and build an adjacency graph with *RegN* nodes, where each node is a register (e.g., the top of Figure 6(d)). Since the live ranges have been assigned registers and each register field must contain one of the *RegN* registers, the adjacency graph should consist of *RegN* nodes only. We then check the permutations of the *RegN* registers and calculate the cost from the adjacency graph to find better solutions. For example, in Figure 6(d), we permute the register numbers and get a solution with cost zero.

However, getting the optimal solution for differential remapping is actually a NP-complete problem, as proved in Theorem .

THEOREM . *Optimal differential remapping is NP-complete.*

PROOF. We prove the theorem in two steps. During the first step, we prove the NP-completeness for *DiffN* = 2 by reducing a known NP-complete problem to it. Then we extend to cases when *DiffN* > 2.

We first briefly introduce a well-known NP-complete problem called the *Traveling Salesman problem (TSP)*. This problem can be stated as follows: Given n cities $C_0, C_1 \ldots C_{n-1}$, and the distance between two cities C_i and C_j as d_{ij}, find the minimal-distance closed path which visits each city once and only once.

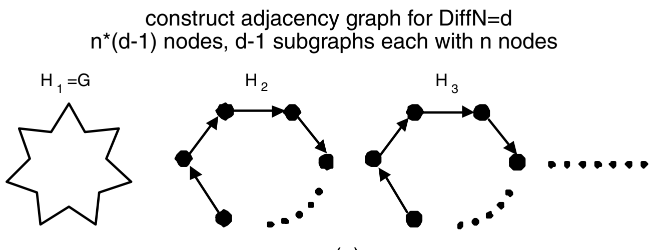
Thus, we reduce TSP to optimal differential remapping. For an instance of TSP, we construct an adjacency graph with n nodes $A_0, A_1 \ldots A_{n-1}$ which correspond to the n cities. Next, we pick a sufficiently large number M. Here, M is larger than all distances between cities. The edge weights on the adjacency graph are assigned as follows. For two nodes A_i and A_j, both edges $\langle i, j\rangle$ and $\langle j, i\rangle$ have weight $M - d_{ij}$, namely, $w_{ij} = w_{ji} = M - d_{ij}$. With this adjacency graph, we set $RegN = n$ and $DiffN = 2$, then the optimal differential remapping solution leads to an optimal solution for the TSP problem. We derive the TSP solution as follows. First, a closed path on the adjacency graph is identified after solving the optimal differential remapping problem, which goes through $node(0), node(1) \ldots node(n-1)$ and back to $node(0)$. Here $node(k), k \in [0, n-1]$ represents the node (or register) that is remapped to register number k. As mentioned earlier, there is a one-to-one correspondence between nodes on the adjacency graph and cities. Therefore, this closed path on the adjacency graph corresponds to a closed path among cities for TSP.

Notice that we assume $DiffN = 2$, and an edge weight w_{ij} is only counted (i.e., satisfies condition (3)) when $(reg\_no(v_j) - reg\_no(v_i)) \bmod RegN = 1$. In other words, only the edge weights on the path $node(0), node(1) \ldots node(n-1), node(0)$ are counted and maximized. Also notice that the optimization objective in Section 4 is to minimize edge weights that do not satisfy condition (3), which is equivalent to maximizing the edge weights that satisfy condition (3) because the sum of all edge weights on the graph is fixed. Also, the edge weights on the adjacency graph correspond to the negated distances between cities, namely, $w_{ij} = w_{ji} = M - d_{ij}$, thus maximizing edge weights on the closed path is equivalent to finding the minimal-distance closed path for TSP.

Next, we prove for more general cases when $DiffN > 2$. Since it has just been proved that optimal differential remapping for $DiffN = 2$ is NP-complete, we reduce this NP-complete problem to optimal differential remapping for $DiffN = d > 2$.

For an instance of optimal differential remapping with $DiffN = 2, RegN = n$, and adjacency graph G, we construct a problem for $DiffN = d$ as follows. There are $RegN = n \times (d-1)$ nodes on the adjacency graph H. The adjacency graph H contains $d-1$ disjoint subgraphs $H_1, H_2 \ldots H_{d-1}$. Among them, H_1 is equivalent to G. For $H_2 \ldots H_{d-1}$, each subgraph contains n nodes and these n nodes are connected clockwise in a circle, that is, each node has two neighbors. The adjacency graph is illustrated in Figure 7(a). All edges on $H_2 \ldots H_{d-1}$ are assigned very high weights. In other words, they should be covered at highest priority. The problem is to find a register assignment scheme that assigns $RegN = n \times (d-1)$ register numbers to the $RegN = n \times (d-1)$ nodes to get the maximal weight coverage.

Assume the register numbers are $0, 1 \ldots n \times (d-1) - 1$. We can categorize them into $(d-1)$ groups, as shown in Figure 7(b). In other words, each group contains an n-number sequence with step d. Notice that due to high edge weights on $H_2 \ldots H_{d-1}$, the optimal assignment should try to cover these edges before covering edges on H_1. Clearly, to cover all edges on $H_2 \ldots H_{d-1}$, we must assign one distinct group of register numbers to each subgraph in $H_2 \ldots H_{d-1}$. In other words, there will be only one group of register numbers (e.g., group 0) left for H_1. Therefore, the optimization objective becomes to cover H_1 with n register numbers, whereas the difference between any two register numbers is a multiple of $d-1$. This is actually equivalent to the $DiffN = 2$ problem we intend to reduce because when $DiffN$ equals d, an edge is only covered when the register


construct adjacency graph for DiffN=d
n*(d-1) nodes, d-1 subgraphs each with n nodes
H_1=G
H_2
H_3

(a)

group 0:	0, (d-1), 2(d-1) ....
group 1:	1, (d-1)+1, 2(d-1)+1....
group 2:	2, (d-1)+2, 2(d-1)+2....
....	
group d-2:	d-2, (d-1)+d-2, 2(d-1)+(d-2)....

(b)

Fig. 7. Illustration for constructing adjacency graph with $\mathit{DiffN} = d$.

numbers of the two end-nodes are consecutive in the sequence of a group. Thus, we have proved the problem is NP-complete for any $\mathit{DiffN} \geq 2$. □

An exhaustive search algorithm has to try all *RegN*! permutations. With the Modulo Equivalence lemma, it can be reduced by a factor of *RegN*, thus a total of $(\mathit{RegN} - 1)!$ permutations need to be checked with edges on the adjacency graph to find their costs. Therefore the complexity of the exhaustive search is $O(\mathit{RegN}^2(\mathit{RegN} - 1)!)$. This is actually tractable for small *RegN* values, but can cost long compilation time when *RegN* gets bigger, for example, 32.

Given the high expense of exhaustive search, we suggest a polynomial-time heuristic algorithm, as shown in Figure 8. This is a greedy algorithm which finds a path in the solution space that achieves the fastest cost reduction until a local minimum is reached. We first define register vector RV as a vector with *RegN* elements. The register vector represents a permutation of the *RegN* numbers from 0 to $\mathit{RegN} - 1$. Given the register vector, we can have the permutation P as $P(i) = RV[i]$, where $i \in [0, \mathit{RegN} - 1]$. Initially, the RV is set to $\langle 0, 1, 2 \ldots \mathit{RegN} - 1\rangle$, which means that the register numbers are not permuted and have one-to-one correspondence to the nodes on the adjacency graph. The algorithm contains a main loop. During each iteration, we attempt all possibilities to swap two elements in the register vector. After each swap, the new register vector is used to find the new cost on the adjacency graph. Only that swap yielding the biggest cost reduction is picked during a given iteration. Therefore, after each iteration, the cost is reduced. When no further cost reduction results, we have found a local minimum and the search stops immediately. To improve solution quality, we can also select a number of (1,000) initial RVs to start with and pick the best solution amongst those generated.

The complexity of the algorithm includes two terms multiplied together. During each iteration in the main loop, $\mathit{RegN} \times (\mathit{RegN} - 1)/2$ pairs can be swapped and checking the cost on the adjacency graph requires $O(\mathit{RegN}^2)$. Finally, each iteration must reduce the cost somehow, therefore we would have O(sum of

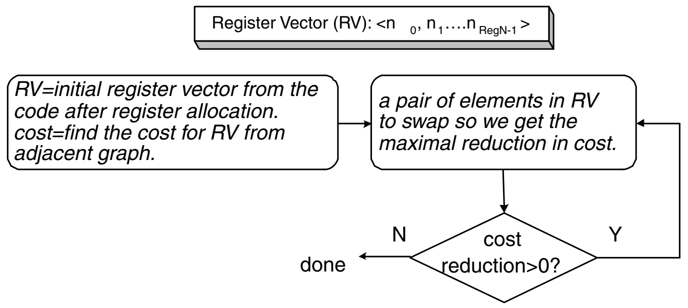

Register Vector (RV): <n_0, n_1....n_RegN-1 >
RV=initial register vector from the code after register allocation. cost=find the cost for RV from adjacent graph.
a pair of elements in RV to swap so we get the maximal reduction in cost.
cost reduction>0?
N
Y
done

Fig. 8. A heuristic algorithm for differential remapping.

weights on the adjacency graph) $\times$ $O(RegN^4)$ as the complexity of the algorithm. Notice that *RegN* is a fixed number for a particular machine, since we always work on an adjacency graph with *RegN* number of nodes, regardless of the number of instructions in a procedure. The complexity is actually a constant. It can easily scale to very big procedures.

6. DIFFERENTIAL SELECT

Differential remapping can be conveniently employed after any register allocator, however, its effectiveness is quite limited. The reason is that the adjacency graph consists of registers and not live ranges, which makes it very dense (in terms of node degrees) and restrictive. For instance, if one node has more than *DiffN* neighbors with different register numbers assigned, no permutation could satisfy condition (3) for all edges incident to these neighbors. Thus, in this section, we attempt our optimization in the select stage of a graph-coloring register allocator, hoping to get a better solution by working on an adjacency graph consisting of *live ranges* instead of registers.

For a graph-coloring-based register allocator, the select stage is in charge of picking register numbers (or colors) for the nodes (live ranges) when adding them back to the interference graph. When picking register number for a node, we need to check the register numbers that have been used by its neighbors on the interference graph. If its neighbors have used all the colors (*RegN* colors), we have to spill the live range. On the other hand, if there are colors unused by its neighbors, the new node can be colored with one of them. If there are multiple unused colors available, most algorithms just pick a color arbitrarily. In Briggs et al. [1994], the authors proposed “biased coloring” to select colors that can potentially reduce extra spills, that is, try to color the source and destination operands of a move instruction with the same color. In this article, we also intentionally pick colors that can benefit from the differential encoding.

In a graph-coloring register allocator, nodes are popped from a stack and inserted into the (partial) interference graph (consisting of nodes popped earlier), while the differential select algorithm as shown in Figure 9 inserts nodes into both the interference graph and adjacency graph. Edges from the new node to those previously inserted are recovered. Next, we determine if the currently popped node should be spilled. If it must be spilled, the register allocator will

Differential Select

1. *Pop a live range from the stack.*
2. *Insert it into both interference graph and adjacent graph.*
3. *Recover the edges to the neighbors that have been inserted earlier than this node.*
4. *Check with the interference graph to get a number of register numbers (colors) that are not used by its neighbors. If no available register number (color), the register allocator will handle that with spills.*
5. *For each allowed register number, find extra cost that will be incurred if the node is assigned this register number.*
6. *Pick the one with minimal cost.*

Fig. 9. Differential select algorithm.

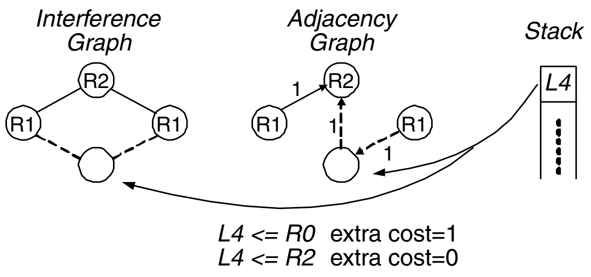


Fig. 10. Example for differential select.

handle this as usual, otherwise we get a number of register numbers (colors) that can be safely assigned to this node. At this point, picking any register number is acceptable for the constraints on the interference graph, therefore we only need to look at the adjacency graph. We then calculate the extra cost of allocating each of the available colors to the node based on condition (3). Finally, we simply pick the register number that incurs the minimal cost.

Figure 10 shows an example. We assume there are three physical registers $R0$, $R1$, $R2$ but the register field is only one bit. When $L4$ is popped off the stack, it can be assigned either $R0$ or $R2$ without interference. However, assigning $R0$ to $L4$ will lead to one extra cost on the adjacency graph. Therefore we should assign $R2$ to $L4$.

Notice that integrating differential select to a graph-coloring-based register allocator does not increase its asymptotic complexity because the complexity is the same for checking either the interference graph or adjacency graph.

7. DIFFERENTIAL COALESCE

In this section, we discuss how to apply differential encoding to an optimal spilling register allocator [Appel and George 2001]. The register allocator uses the ILP solver to find optimal spilling at the first step. Optimal spilling is achieved by enforcing that at each program point, at most *RegN* live ranges are colive. This gives an optimal number of spills because we can always assign registers to live ranges if enough move instructions are inserted. In the worst

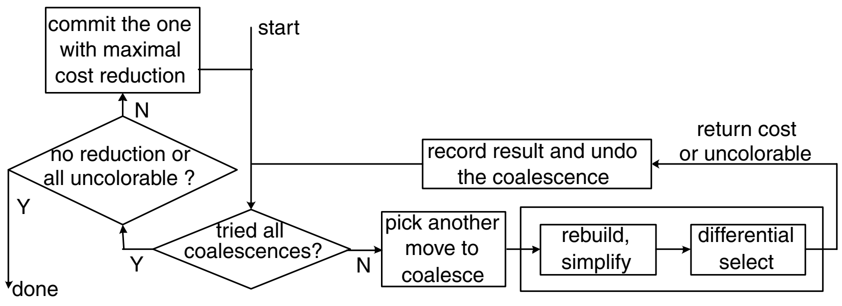

commit the one with maximal cost reduction
start
N
no reduction or all uncolorable ?
Y
done
record result and undo the coalescence
return cost or uncolorable
tried all coalescences?
Y
N
pick another move to coalesce
rebuild, simplify
differential select

Fig. 11. Differential coalesce algorithm.

case, there will be a large number of moves inserted to split the live ranges. The second step of the register allocator uses a heuristic algorithm to coalesce moves and color the interference graph. They first do this aggressively, as in Chaitin et al.'s [1981] register allocation, then undo some coalescences once the graph becomes overconstrained, that is, becomes uncolorable without introducing spills.

A flow graph of the differential coalesce algorithm is shown in Figure 11. It coalesces nodes step-by-step and every time it tries to pick a pair of nodes for maximal cost reduction during coalescing. Notice that the "rebuild and simplify" and "differential select" stages have been made into functions which get called and return the cost on the adjacency graph, or return "uncolorable" if the graph becomes uncolorable after the current coalescence. The result of each coalescence attempted is recorded, but the coalescence itself is not committed. We undo the coalescence and try another until all possibilities have been exhausted on the current graph. Then we apply the one with maximal cost reduction. Here, the cost not only includes that for differential encoding (calculated from the adjacency graph) but also includes the cost eliminated due to move instructions removed. As we have observed, the cost reduction is actually more sensitive to the reduction of the differential encoding cost. Besides, we assume that a *set_last_reg* instruction is of the same computation cost as a move instruction, since the former also moves a value to an on-chip register. In case either there is no cost reduction or all coalescences lead to uncolorable graph, the algorithm terminates. By considering the cost due to both move and *set_last_reg* instructions, the algorithm targets the overall optimization objective.

The complexity of the algorithm is higher than the original, which coalesces aggressively first instead of trying all possible coalescences and picking the best candidate at each coalescence. Theoretically, we have the complexity as $O((\#\text{moves})^2) \times O(\text{complexity of differential select})$. Here the subroutine consisting of "rebuild and simplify" and "differential select" is invoked at most $O((\#\text{moves})^2)$ times, and the complexity of "differential select" is much higher than "rebuild and simplify". However, since differential coalesce mainly works with the second step of the optimal spilling register allocator, while the majority of the compilation time is spent on the first step due to the ILP solver, the compilation overhead is actually very small, as indicated in the result section.

8. ADAPTIVE DIFFERENTIAL REGISTER ALLOCATION

In this section, we discuss ways to apply differential register allocation more flexibly. From previous sections, we know that differential register allocation does not come free of cost. If the number of architected registers is enough with direct encoding, applying differential register allocation might unduly add a large number of extra instructions, which not only bloats code size but suffocates instruction decode and I-cache. On the other hand, differential register allocation is less effective when encoding space is ample or the register pressure low. Therefore, it fits better for low-end processors. However, even for processors with ample architected registers, register pressure can increase due to compiler optimizations like aggressive inlining, allocating global variables, promoting aliased variables, etc. It is likely that in some regions register pressure is very high, typically in those frequently executed and heavily optimized code segments. In such cases, differential register allocation is desirable.

Therefore, it is important that differential encoding is applied properly such that the benefit maximally exceeds the cost incurred. We would expect that differential encoding can be easily turned on and off. In other words, we only need to enable differential encoding when it is beneficial. Besides, it would be helpful if we could tune its parameters according to the register pressure.

8.1 Adjust the *RegN*/*DiffN* Ratio

We will show in Section 12 that the *RegN*/*DiffN* ratio affects the numbers of spills, moves, *set_last_reg* instructions, and the performance. Once the size of the register field is fixed by the ISA designer, both *DiffW* and *DiffN* are fixed. Changing *RegN* or the *RegN*/*DiffN* ratio has its implication tied to the register pressure. Intuitively, if the register pressure is low, we should either disable differential encoding or set the *RegN*/*DiffN* ratio low because increasing the number of architected registers cannot result in much benefit. Meanwhile, condition (3) is less likely to be satisfied with a high *RegN*/*DiffN* ratio. In other words, the high ratio will incur the cost of more *set_last_reg* instructions being inserted. If the register pressure is high, we should set a high *RegN*/*DiffN* ratio. This immediately gives us more architected registers, which is directly translated to spill reduction. Since spills are generally much more expensive than moves and *set_last_reg* instructions, a high *RegN*/*DiffN* ratio is expected to improve the overall performance.

To facilitate adaptive differential register allocation, we introduce a new instruction as follows.

$$\begin{aligned} &\mathit{set\_RegN}(\mathit{value})\text{: set } \mathit{RegN} = \mathit{value} \text{ if } \mathit{value} \neq 0 \\ &\qquad\qquad\qquad\qquad\qquad\qquad \text{else disable differential decoding.} \end{aligned}$$

The *set_RegN* instruction contains one parameter. If $value \neq 0$, it tells the instruction decoder that a new *RegN* value should be used. If $value = 0$, the decoder switches to direct decoding. As soon as another *set_RegN* instruction is encountered in the instruction stream with $value \neq 0$, the decoder recommences differential decoding. Thus, the *set_RegN* instruction allows us to conveniently disable/enable the differential encoding mode and adjust the *RegN* value. As mentioned in Section 2.1, the modulo operation can easily handle different *RegN*s with low hardware expense.

8.2 *RegN* Regions

By inserting *set_RegN* instructions, the decoder can decode differently at different program points. Clearly, we cannot insert *set_RegN* instructions arbitrarily. At any given program point, the *RegN* value must be uniquely determined regardless of the path taken to this program point, otherwise the register number after decoding could be different. Consequently, the control flow graph can be split into a number of regions called *RegN Regions*.

Definition 3 (*RegN Region*). A *RegN* region is a maximal connected program segment on the control flow graph which assumes the same *RegN* value (or no differential encoding). It contains connected parts from several basic blocks. The boundary of a *RegN* region should be either *set_RegN* instructions or entry/exit points.

According to the definition of *RegN* region, *set_RegN* instructions must be properly inserted for a consistent *RegN* value in the region. Although splitting the control flow graph into *RegN* regions makes differential encoding more flexible to apply, we have to address questions such as how to set the boundary of the regions, what should the $RegN/DiffN$ ratio be, how to make connection at the boundary points, etc. Next, we give two examples to illustrate the pertinent issues.

In Figure 12, we give an example illustrating how region splitting could help reduce the cost. In addition, we can shift register numbers to reduce move instructions on the region boundary.

Figure 12(a) shows three live ranges $L1$, $L2$, $L3$ in a piece of sequential code. We can roughly split the code into two regions with different register pressures. In region 1, only $L1$ and $L2$ are live, while all three are live in region 2. In other words, live ranges $L1$ and $L2$ live across regions. The adjacency graph in the two regions are shown on the right side of Figure 12(a). Also, we set $DiffN = 2$, in other words, if more than two architected registers are needed to avoid spills, we have to use differential encoding.

Notice that we must use differential encoding in region 2 to avoid spills. But should we merge both regions and set $RegN = 3$ or encode differently for the two regions? If they are merged, we save one *set_RegN* in between the two regions. However, the cost is actually increased. The adjacency graph in Figure 12(a) shows that we must have $(reg\_no(L2) - reg\_no(L1)) \bmod 3 = 2$ to cover both edges on the adjacency graph of region 2, that is, the register numbers for L2 and L1 must be two hops away clockwise on the circle, as shown in Figure 1. One of the possible assignments is shown in Figure 12(b). However, the two edges on region 1's adjacency graph cannot both be covered. This means 3 *set_last_reg* instructions are required in this region to handle out-of-range accesses.

In Figure 12(c), we disable differential encoding in region 1 and set $RegN = 3$ in region 2. Since differential encoding is disabled in region 1, we can completely disregard the adjacency graph. Although all edges are covered on region 2, we

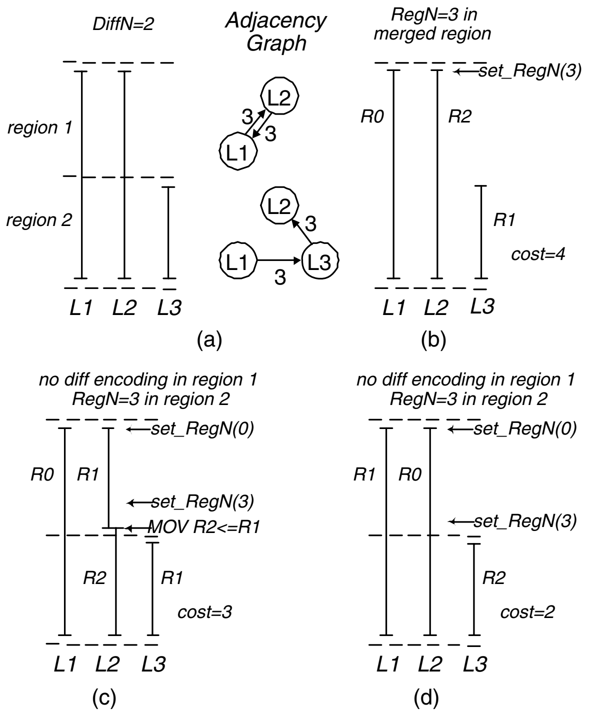


Fig. 12. Cost reduction through region splitting.

must make the register numbers consistent for $L1$ and $L2$ in both regions. Unfortunately, only $R0$ and $R1$ can be used in region 1. Thus we need a move instruction $MOVR2 \leftarrow R1$ at the boundary of the two regions to rename the register number for $L2$. Due to the move instruction, the adjacency might be changed and we need to take this into consideration, slightly complicating matters. If we assume the adjacency graph is not changed, the total cost is three, counting two *set_RegN* instructions and one move instruction. Although this is better than merging the two regions, we can actually further reduce the cost. According to the Modulo Equivalence lemma (Section 5), we can shift the register numbers for the three live ranges without increasing the cost in region 2. After proper shifting, we get the register allocation scheme in Figure 12(d). Only the two *set_RegN* instructions are needed, reducing the cost to two.

However, splitting code into *RegN* ranges is not always beneficial because we need to insert *set_RegN* instructions and sometimes moves at the boundaries for live ranges that cross region boundaries, as well. We show such an example in Figure 13. In Figure 13(a), live ranges $L1$ and $L2$ span both regions 1 and 2, whereas live range $L3$ only stays in region 2. The register pressure in region 1 is low enough to avoid the use of differential encoding, while in region 2, *RegN* should be at least 3 to avoid spills. If the two regions are not merged as in Figure 13(b), no solution exists to cover all edges on the adjacency graphs of both regions. This leads to two *set_RegN* instructions being inserted. Actually, it would be a better choice to simply merge the two regions because $L1$ and $L2$

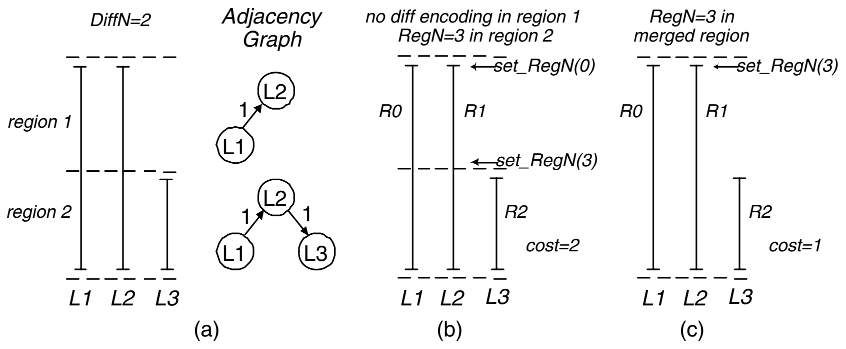


Fig. 13. Cost reduction through region merging.

get the same register number in both regions and the edges on the adjacency graph are covered anyway. As in Figure 13(c), merging the two regions reduces the cost to only one *set_RegN* instruction.

The preceding examples illustrate the intricacy of forming *RegN* regions such as to minimize cost. These examples tell us a several things:

— Registers should first be allocated independently in each region and *RegN* values are determined based upon the optimization objective. Roughly, this should be set to match the register pressure to avoid generating too many spills, but also control the number of *set_last_reg* instructions.

— At region boundaries, *set_RegN* instructions are needed to change the *RegN* value. Also, we might need to insert moves to rename live ranges that cross regions. In other words, live ranges that span multiple regions must be properly handled to reduce renaming cost.

— The first example shows that sometimes splitting one region into two helps to cover edges on both adjacency graphs. In addition, shifting register numbers based on the Modulo Equivalence lemma could remove some of the moves at the boundary points. The second example implies that merging is favored when adjacency graphs are properly covered with a unified *RegN*. By raising *RegN* for one region, we can merge it with its neighbor with higher *RegN*. As long as raising *RegN* does not incur more cost than the reduction of *set_RegN* and move instructions at the boundary, merging is preferred.

Figure 14 shows the algorithm flow graph for adaptive differential register allocation. At the beginning, we assume the CFG has been split into many small regions called unit *RegN* regions. Each unit *RegN* region is actually a single instruction. Then the heuristic algorithm starts to merge these *RegN* regions greedily. The algorithm is cost-driven in that during each step, that pair of regions are merged which can lead to the biggest cost reduction. However, a full-fledged step to determine the cost for each possibility would involve too many calculations. Therefore we first attempt all mergers that can be quickly carried out. The fast merger is to merge nearby regions that are at low register pressure. As long as the merged region does not incur spills with direct

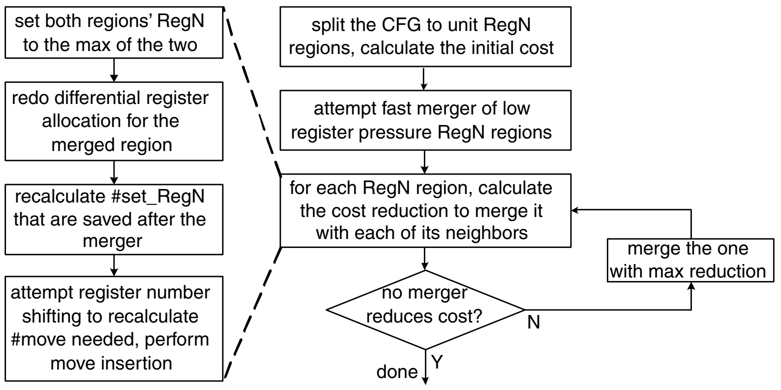

set both regions' RegN to the max of the two
redo differential register allocation for the merged region
recalculate #set_RegN that are saved after the merger
attempt register number shifting to recalculate #move needed, perform move insertion
split the CFG to unit RegN regions, calculate the initial cost
attempt fast merger of low register pressure RegN regions
for each RegN region, calculate the cost reduction to merge it with each of its neighbors
merge the one with max reduction
no merger reduces cost?
N
Y
done

Fig. 14. Algorithm for adaptive differential register allocation.

encoding, we merge them immediately. In this manner, regions with low register pressure are quickly formed, which greatly improves the efficiency of the algorithm.

Next, we calculate the cost for merging two neighboring regions. The calculation of cost reduction takes four substeps, as shown on the left side of Figure 14. From previous examples, we need to consider cost change both inside the two regions and at the boundary. First, that region with lower *RegN* changes its *RegN* to be equal to the bigger of the two regions; if their *RegN*s are the same, we can skip this substep. We then perform differential register allocation in the newly merged region with one of the three approaches proposed in previous sections. After the internal cost change is known, we look at the number of *set_RegN*s that are saved due to the merger. This can be easily found by checking the region with new *RegN* and its neighbors. The final substep is to decide how many move instructions should be put at the boundary points. As shown in Figure 12, shifting register numbers might reduce the renaming required. We attempt to shift register number for the newly merged region to see its assigned register numbers can better match its neighbors. Finally, since inserting moves could affect the access sequence at the boundary points and induce *set_last_reg* instructions, we patch the code and add this cost as well. The entire loop continues until we cannot find any merger that is profitable. The complexity of the algorithm is mainly determined by the loop to find the best pair to merge in each step, since the fast merger merely quickly merges low register pressure regions. Among the four substeps, redo differential register allocation is the most computationally intensive. Therefore, assuming that there are M regions after the fast merger, we get the complexity as $O(M) \times O(M^2) \times$ (maximal complexity of differential encoding among all regions). Here, the first term is the maximal number of actual mergers that can be conducted, that is merged to one region eventually. The second term is for the number of region pairs checked before committing that with the maximal cost reduction. The last term stands for the maximal complexity for the substep to do differential encoding for the merged region.

9. AUXILIARY OPTIMIZATION TECHNIQUES

There are a number of techniques we can use to change the adjacency graph without affecting the program semantics. In this section, we talk about two optimization techniques, namely, instruction scheduling and algebraic transformation. The former changes access the sequence between instructions and the latter changes the access, sequence within an instruction.

Instruction scheduling changes the access sequence and edge connections on the adjacency graph. Figure 15(a) shows an example. The original code on the left has three edges on the adjacency graph and the total weight is three. Simply reordering the first two instructions can reduce one edge and the total weight by one. Notice that for the original code, the two edges between node $L1$ and $L2$ enforce that the two nodes must be assigned with the same register number to avoid extra cost. This is a very strict restriction. Due to the interference between $L1$ and $L2$, they cannot actually be assigned to the same register. Therefore extra cost is inevitable in this case. Since there is no dependency between the first two instructions, this transformation is completely free and legal, but generates an adjacency graph with less restriction.

Algebraic transformation based on algebraic laws such as associativity, distributivity, and commutativity can change the access sequence inside an instruction if such a transformation is allowed. A simple example is demonstrated in Figure 15(b). The two-line code on the left of the figure leads to an adjacency graph with two edges between the two live ranges and a total weight of three. Similarly, due to the interference between the two live ranges, the two edges cannot both be covered. However, applying the commutativity law to the second instruction gives us an adjacency graph with only one edge and the total weight of one.

From these examples, we observe that if edges can be converted into self-looped edges, namely, edges which loop back to the same node, they will not appear on the adjacency graph. For instance, in Figure 15(a) one edge becomes $L1 \rightarrow L1$, and in Figure 15(b) we have $L2 \rightarrow L2$ and $L1 \rightarrow L1$ after conversion. Thus both the total weight and edge number are reduced. In addition, as a rule of thumb, we would like to reduce the number of edges even if the total weight is not reduced. In other words, having one edge with weight two is probably better than having two edges with weight one. This comes from our observation that a lesser number of edges most likely incurs less cost.

Currently, the two techniques are implemented in a simple manner before register allocation. For instruction scheduling, we check instructions in the same basic block. A dependency DAG is constructed where each instruction becomes a node on the DAG. We attempt all instruction scheduling possibilities without violating the dependencies to get the one that leads to the minimal cost on the access graph. For algebraic transformation, this is only attempted for each individual instruction. We try all legal transformations for the instruction and follow that which is best on the access graph.

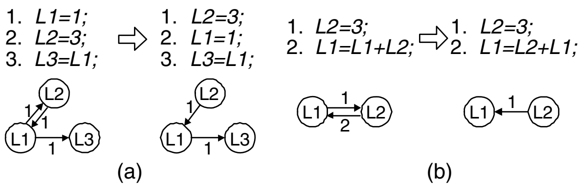


Fig. 15. Illustration of two auxiliary optimizations.

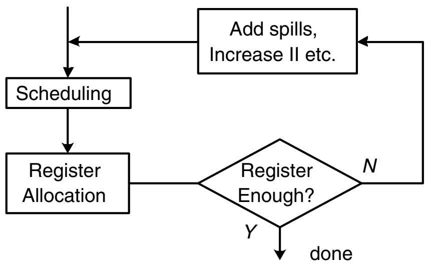


Fig. 16. Modulo scheduling flowchart.

10. APPLICATION TO SOFTWARE PIPELINING

Software pipelining attempts to extract parallelism out of loops by overlapping the execution of several consecutive iterations. The overlapping of iterations could impose high register requirements. Furthermore, if loop optimizations like "unroll and jam", "subexpression elimination", and "back substitution" are applied as well, the register pressure could be further increased. Although hardware-managed rotating registers (e.g., in the Itanium processor) could help to reduce register pressure, they are not always available. On the other hand, compile-time renaming through modulo variable expansion (MVE) [Lam 1987] has to unroll the loop kernel, leading to higher register pressure. As indicated in Zalamea et al. [2000a], a significant portion of the execution time is spent on loops requiring more than 64 registers. There are many register allocation schemes proposed with respect to modulo scheduling, such as Llosa et al. [1996], Rau et al. [1992], Ruttenberg et al. [1996] and Wang et al. [1994]. Typically, the algorithms follow the flowchart in Figure 16. The register allocation algorithm does not introduce spills because spills occupy memory units, thus the spill decision must be made together with scheduling. Due to the relatively small number of instructions in a loop, the register allocator is normally highly optimized. As a result, we propose to apply differential remapping only, such that our scheme does not affect the original register allocator. Notice that *set_last_reg* instructions only serve for decoding, that is, set the *last_reg* when decoding register operands. After the decode stage, these instructions are removed and do not get into the execution stage. Therefore inserting such instructions will not affect the original schedule that is executed. To avoid disrupting the original instruction word, which typically contains multiple instructions, we can choose to group *set_last_reg* instructions together or promote them before the module scheduled code. By setting the delay number properly, all instructions can be differentially decoded properly.

11. OTHER CONSIDERATIONS

11.1 Register Classes

Normally, registers belong to multiple classes such as integer registers, floating point registers, etc. Our approach is still applicable under such circumstances. We can categorize register accesses based on their classes and perform encoding and decoding separately. During encoding, the access sequence only contains registers belonging to the same register class. In other words, if we start to encode integer registers, accesses to other classes are skipped. During decoding, we need a separate *last_reg* register for each class. Therefore, all register allocation algorithms can be applied accordingly to each class of registers.

11.2 Special-Purpose Registers

Some of the registers may have special purposes, such as stack pointer, frame pointer, zero register, etc. Such registers might be used so frequently that the nodes on the adjacency graph become overconstrained. Thus, it is not beneficial to include these registers in differential encoding, which could induce a large number of *set_last_reg* instructions because we will frequently see these registers being accessed after other registers. These registers must still be properly addressed. Remember that we have *DiffW* bits for encoding. In practice, we can reserve some register numbers in the encoding space for the special-purpose registers, therefore $DiffN < 2^{DiffW}$. For both encoding and decoding, these register numbers are assumed to be encoded directly (may add by a constant value), for instance, if there are sixteen physical registers $R_0 \ldots R_{15}$, and R_{15} is stack pointer. To encode into three bits, we can reserve number 7 for the stack pointer. Then *DiffN* actually equals 7, namely, only 0.6 are used for differential encoding. Whenever we see 7 in the register field, it is the stack pointer. We add it by 8 to get R_{15}.

11.3 Context Switches, Function Calls

During context switches, only the *last_reg* should be stored together with the context. This overhead is almost negligible. Function calling conventions typically designate a number of registers as caller-saved/callee-saved. For approaches (2) and (3), they are just part of the register allocator, therefore they won't affect the way the register aclloctor handles the calling convention (typically, these registers are precolored, etc.). For differential remapping, if all allocated registers are remapped, the caller-saved/callee-savee registers might be remapped to other registers, breaking the calling convention. This problem can be solved as follows. We first apply differential remapping regardless of the caller-save/callee-save conventions, then remedy them separately by inserting a few *set_last_reg* instructions, for example, if the caller-save registers are $r4, r5, r6$, and are remapped to $r8, r0, r1$. There should be store instructions in front of a function call. They originally looked like ST $r4$..; ST $r5$..; ST $r6$.. and now become ST $r8$..; ST $r0$..; ST $r1$.. We can simply restore them to the original register numbers and then apply differential encoding, that is, we may insert a few *set_last_reg* instructions in the middle of these caller-save instructions if differences are out of range.

11.4 Other Alternatives

In addition, there are other interesting alternatives that can be further evaluated. For example, we can change the *last_reg* per instruction instead per register field. Also, the access order can be more flexible. Different types of instructions may have different access order. Since access order affects the order of registers in the access sequence and the edge connection on the adjacency graph, a more flexible one may incur less cost. All these could be topics of our future work.

11.5 Work with an Instruction Scheduling Phase

Instruction scheduling phases are frequently seen in optimizing compilers. Here, we talk about instruction scheduling phases in general, instead of the one mentioned in auxiliary optimizations which is integrated with our register allocation. Differential select and differential coalesce are part of the register allocation pass inside the compiler. In other words, our optimization has been integrated with register allocation, therefore instruction scheduling can be applied either before or after. If instruction scheduling is applied afterwards, part of the access sequence might be changed. To remedy this, the instruction scheduling pass could consider differential encoding as in our register allocation pass. This does deserve further study on how to properly integrate them. Alternatively, differential remapping is a complete postpass optimization. It can be applied after both register allocation and instruction scheduling when the access sequence has been determined. Finally, with differential encoding, *set_last_reg* instructions only exist until the decode stage; they will not enter the pipeline and affect scheduled code.

12. EVALUATION

We evaluate our schemes for two types of machine configuration using the Simplescalar toolset [Burger and Austin 1997]. As mentioned earlier, differential register allocation is only profitable when the benefit of more architected registers exceeds the cost due to extra instructions. By combining it with differential register allocation, it works well with a low-end processor with tight encoding space. The benefits decrease as more registers are supported on the architecture. Under certain circumstances, high-performance processors can also take advantage of this technique by applying it adaptively for high register pressure regions, such as software pipelined code.

12.1 Evaluation on a Low-End Processor with Tight Encoding Space

We evaluate our approaches on a machine model similar to the ARM/THUMB architecture. It is a five-stage in-order issue processor, detailed in Table I. Both I-cache and D-cache are 32-way 16KB, while there is no level-two cache present. The cache miss penalty is 64 cycles. As mentioned earlier, most THUMB instructions can only see eight registers since the register field is only three bits.

Table I. Parameters for Our Machine Model

Pipeline	5-stage	L1 I-cache	32-way 16K 32B block
Issue	2 issue, in-order	L1 D-cache	32-way 16K 32B block
#Registers	Up to 16	L1 Latency	Hit 1 cycle, miss 64 cycles

Therefore we assume the original ISA only allows eight registers to be addressed, but the architecture actually supports sixteen registers. We assume that decode latency gets one cycle longer due to the extra hardware. We evaluate twelve benchmark programs from Mibench [Guthaus et al. 2001]. Mibench is a benchmark suite close to the industrial standard EEMBC benchmark for embedded processors. Due to library problems, not all benchmarks could be compiled and run successfully. Among those successful we pick those with relatively large sizes because the real-world applications running on embedded processors tend to be much larger nowadays, such as the applications on WinCE. Moreover, we do not measure power, area, etc., since our main objective is to improve performance.

In this experiment, we replace GCC's register allocation phase by implementing iterated register allocation [George and Appel 1996]. We call this the "baseline" code. The second setup is called "remapping", in which the baseline code undergoes differential remapping as described in Section 5. The third setup, called "select", follows differential select, where the select phase of the iterated register allocator is modified as described in Section 6. To evaluate differential coalesce, we implemented optimal spill register allocation [Appel and George 2001] and modified its coalesce phase as in Section 7.

Please note that differential encoding allows more registers to be used and thus the binaries for "remapping", "select", and "coalesce" are produced with an appropriately higher number of registers than the baseline and optimal spill register allocators. For the three approaches with differential register encoding and allocation, we set $RegN = 12$ for Figure 17 to Figure 20. Thus, the code for "remapping", "select", and "coalesce" can address twelve registers as against eight registers used for the baseline and optimal spill cases. Since there are three bits for each register field and no special-purpose registers are included, we always have $DiffN = 8$ and $DiffW = 3$.

Figure 17 shows comparisons for percentage of static spills generated over the entire code. We observe a dramatic decrease of spills from the baseline for all three approaches, since differential encoding allocates with twelve registers which contribute to the reduction of spills. Differential remapping and select are quite close, indicating that the number of spills is not sensitive to our algorithm, but rather to the available registers and register allocator. For O-spill (optimal spill), although it generates many less spills than the baseline, it is still a little worse than the differential ones (named 'remapping', 'select', and 'coalesce' in the graph) because it allocates with eight registers. On average, the spill code percentages are 10.12%, 6.66%, 6.59%, 7.13%, and 5.42% for "baseline", "remapping", "select", "O-spill", and "coalesce", respectively.

Figure 18 compares the cost for the three differential algorithms. Here cost means the number of *set_last_reg* instructions introduced over the entire code

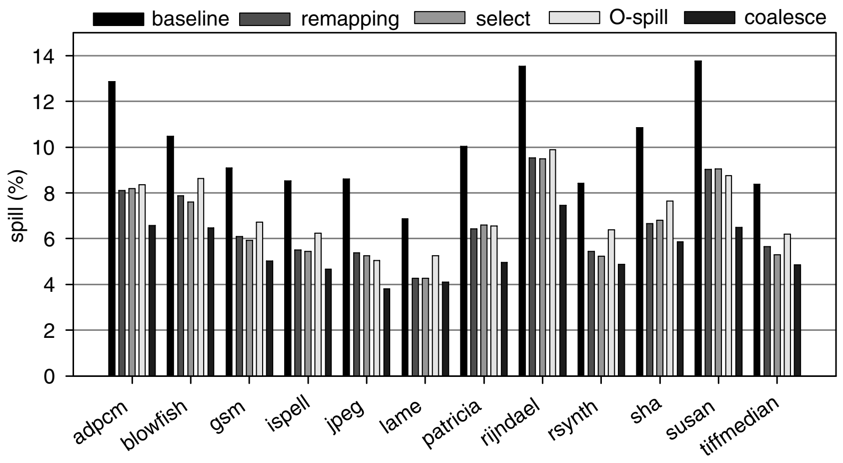


Fig. 17. Spill comparison.

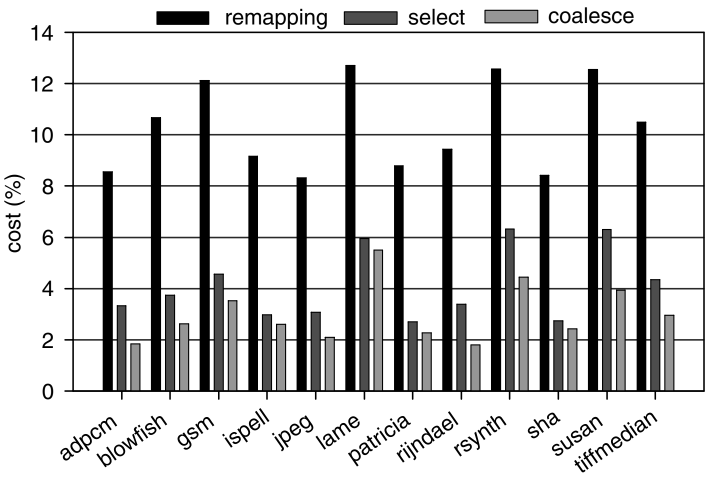


Fig. 18. Cost comparison.

as a percentage. These are static numbers. Not surprisingly, remapping generates a large number of *set_last_reg* instructions, whereas it is much less for the other two approaches. Between the last two approaches, differential coalesce is slightly better, perhaps because the optimal spilling algorithm initially reduces the code growth. Therefore we will see shorter access sequence, thus less weights on the adjacency graph. On average, the percentages for the three algorithms are 10.31%, 4.12%, and 3.01%, respectively.

Figure 19 shows code size comparisons. All numbers are normalized to the baseline. Although differential encoding reduces the number of spills, some of the approaches cause more *set_last_reg* instructions. In short, remapping increases code size by 6.2% over the baseline, whereas differential select only affects the code size by less than 1%. Both "O-spill" and "coalesce" actually

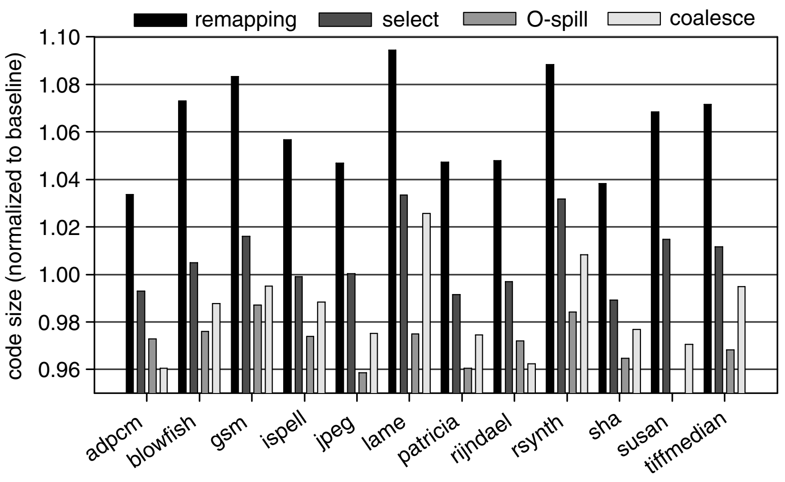


Fig. 19. Code size comparison.

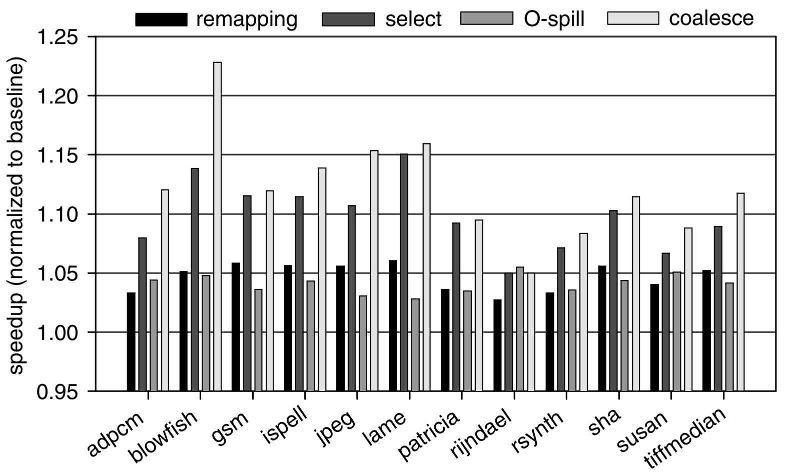


Fig. 20. Speedup comparison.

reduce the code size (3% and 2%, respectively) mainly due to reduction in spill code.

Finally, we look at the speedup of the four cases over the baseline in Figure 20. Due to the high cost incurred for remapping, most of the benefits with more registers are eliminated. Also, coalesce is best due to its reduction in spills and *set_last_reg* instructions. The amount of improvement is quite irregular across benchmarks, perhaps because we rely on static weight estimation instead of profile information. In summary, the three approaches achieve average speedups of 4.6%, 9.8%, and 12.2%, respectively, whereas "O-spill" achieves 4.1% speedup. In other words, "coalesce" is able to improve over "O-spill" substantially by taking advantage of more registers and reducing spills.

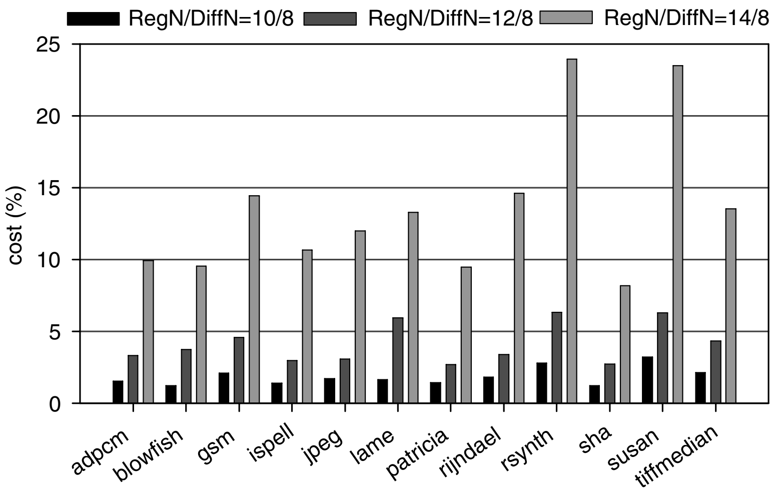


Fig. 21. RegN/DiffN sensitivity test-cost comparison.

12.2 *RegN*/*DiffN* Sensitivity Test

Next we intend to evaluate how performance varies with the *RegN*/*DiffN* ratio. We will continue using the same machine model and configuration as in the previous subsection. A similar study will be conducted for a machine model with more registers in Section 12.3. Without loss of generality (we observed a similar trend for the other two algorithms), we only look at differential select. As mentioned earlier, *DiffN* and *DiffW* are not changed once the register field is fixed, therefore the *RegN*/*DiffN* ratio is basically decided by *RegN*. If *RegN* is too small, more spills may result. On the other hand, an unnecessarily large *RegN* could induce many *set_last_reg* instructions.

From Figure 21, we observe a significant increase in the number of *set_last_reg* instructions. This confirms that if *RegN* is too far away from *DiffN*, the cost can grow significantly and eliminate most of the benefits of having more registers. This is probably due to the larger spectrum of registers that can be used; many of the weights on the adjacency graph cannot be satisfied. In other words, it is more likely that the difference between the two end-nodes on an edge is larger than *DiffN* leading to extra instructions being inserted. The average cost in terms of the percentage of *set_last_reg* instructions over the entire code is 1.85%, 4.12%, and 13.59% for the three cases, respectively.

We observe a similar trend for code growth in Figure 22. This is perhaps because the growth of *set_last_reg* instructions far exceeds the reduction of spills when *RegN*/*DiffN* gets larger. Statistically, when *RegN*/*DiffN* changes from 10/8 to 12/8, the percentage of spills over the entire code size is reduced by about 1%. Meanwhile, Figure 21 tells us the percentage of *set_last_reg* is cut by over 2%. When the *RegN*/*DiffN* ratio is raised from 12/8 to 14/8, it yields only 0.5% spill reduction. On the other hand, *set_last_reg* instructions increase considerably. Consequently, the code size is reduced by 1% for *RegN*/*DiffN* = 10/8, while increasing by 0.69% and 8.7% for *RegN*/*DiffN* = 12/8 and *RegN*/*DiffN* = 14/8, respectively.

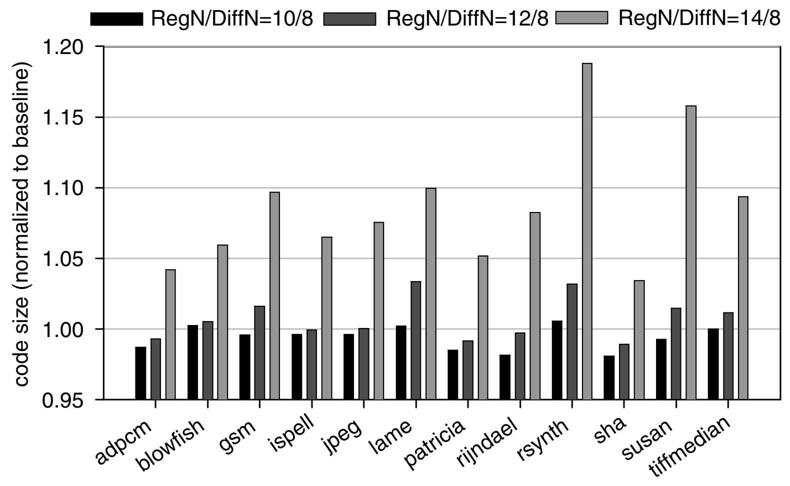


Fig. 22. RegN/DiffN sensitivity test-code size comparison.

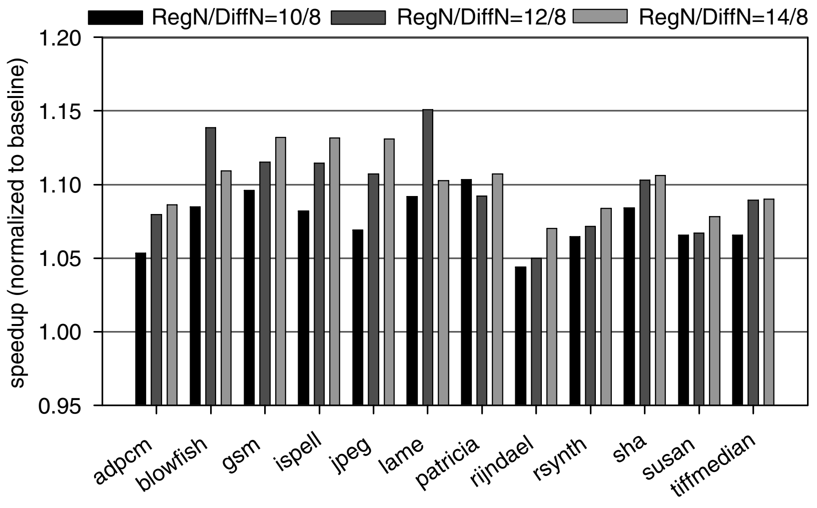


Fig. 23. RegN/DiffN sensitivity test-speedup comparison.

Next we intend to evaluate how performance varies with the number of registers available for differential encoding. In Figure 23, we attempt $\mathit{RegN}/\mathit{DiffN} = 10/8, 12/8$, and $14/8$. One important observation is that with $\mathit{RegN} = 14$ registers, we see mixed speedups over $\mathit{RegN} = 12$. On average, both are near 10% speedup over the baseline, whereas with 10 registers the speedup is only 7.5%. It is understandable that more registers give diminishing returns on speedup, since spill reduction becomes smaller as well. On the other hand, due to the growth of *set_last_reg* instructions, I-cache and decoding may suffer. Our rule of thumb is that *RegN*/*DiffN* should be somewhere between 1 and 2.

12.3 Evaluation on a High-Performance Processor

In this subsection, we evaluate differential encoding on a machine with a higher-performance profile. As mentioned earlier, the register bottleneck only exists in some special cases on high-performance processors. Here we apply it to help with modulo scheduling where register pressure becomes very high after such optimization. Additionally, we will conduct a *RegN*/*DiffN* sensitivity test along with the performance evaluation.

We work with a VLIW (in-order issue) machine model with 32 architected registers and 64 physical registers. There are four functional units and two memory ports. Still, we assume that decode latency is one cycle longer. We adopt the modulo scheduling algorithm as described in Zalamea et al. [2000a]. The scheduling algorithm carefully spills variables when the number of used registers exceeds the number of available registers. Notice that in such cases we can increase the initiation interval (II) to reduce register pressure, which might avoid spills. However, earlier research [Llosa et al. 1996] has suggested that spilling most likely leads to less slowdown.

We studied 1,928 (innermost) loops selected from the SPEC2000 integer benchmark suites. The execution of these loops constitutes over 80% of all execution time. We also assume that other loop optimizations have been enabled in the compiler during the code generation. We found that among these loops, about 11% require more than 32 registers, which means they must incur spills. These loops are typically big and account for a significant portion of the overall loop execution time (over 30%). We set $DiffN = 32$, then change *RegN* to a number of values: 32, 40, 48, 56, 64, that is, the *RegN*/*DiffN* ratio takes a series of values from 1 to 2. When $RegN = 32$, it means no differential encoding is involved, namely, it is the baseline software pipelined code *without* differential encoding.

Notice that we only use differential encoding for those loops demanding more than 32 registers to reduce their spills. For loops with enough registers (no spills), differential encoding is turned off with the *set_RegN* instruction to avoid extra cost due to *set_last_reg* instructions. This could be thought of as a simple adaptive differential register allocation scheme which either disables or enables differential encoding with a fixed *RegN*/*DiffN* ratio for all loops under study (evaluation of the full-fledged adaptive scheme described earlier in the article will be separately discussed shortly). We count the number of *set_RegN* instructions in code growth as well, although they are not involved in loops and their performance impact is marginal.

Table II shows the speedup for the code after our optimization. We can observe significant speedup for the loops that are optimized, as shown in the second column. A larger *RegN* tends to improve the performance a lot, however, the improvement gradually saturates after *RegN* is larger than 48. The speedup for all loops is from 10.23% ($regN = 40$) to 17.24% ($RegN = 64$). The overall speedups are close to those for all loops because most of the cycles are spent on loops.

Table III presents the number of spills in optimized loops and the code growth. Clearly, the number of spills decreases appreciably when *RegN* is

Table II. Speedup Comparison

RegN	%speedup (opti. loops)	%speedup (all loops)	%speedup
(base) 32	0	0	0
40	34.74	10.23	8.46
48	57.32	14.91	12.13
56	70.39	16.10	13.62
64	73.12	17.24	14.38
adaptive	77.48	18.11	14.97

Table III. Spill and Code Growth Comparison

RegN	total #spills	%code growth (opti. loops)	%code growth (all loops)	%code growth
(base) 32	4124	0	0	0
40	1580	−1.66	−0.39	−0.06
48	983	3.16	0.74	0.12
56	906	17.18	4.03	0.65
64	809	29.61	6.95	1.13
adaptive	814	2 .59	0.61	0.099

increased from 32 to 40 and 48. The reduction becomes much smaller after this. The third to fifth columns are about code growth for “optimized loops”, “all loops”, and “all code”, respectively. Although spills are reduced, more costs are induced with larger *RegN*, as mentioned earlier. Also, since we can only apply differential remapping, the number of *set_last_reg* instructions increases a lot. However, the part of loops that are being optimized only takes a small portion of the entire code. Therefore the overall code growth is 1.13% at most. Notice that when $RegN = 40$, we can actually reduce code size because more spills are saved versus the extra cost.

The last rows of both tables are about the adaptive differential scheme, and will be addressed in the next subsection.

12.4 Evaluation for Adaptive Differential Register Allocation

To evaluate adaptive differential register allocation, we experimented with both machine models. The data for the low-end machine model is reported in Table IV, where differential select is set as the register allocation algorithm. The adaptive scheme allows *RegN* to be set differently at different program points and the code to be split into *RegN* regions with different *RegN*s. For this model, we set the largest *RegN* to be 16, while the register field still contains 3 bits, that is, $DiffW = 3$, $DiffN = 8$.

We categorize results into total number of regions (column 2), average region size (column 3), percentage of spills (column 4), percentage of cost (column 5), and the speedup (column 6). The number of *RegN* regions largely reflects code size. For large benchmarks, there could be thousands of regions. Meanwhile, the average region size is mostly in the range of 10 to 30 instructions. Here we only average the regions which use differential encoding. As a matter of fact, only about 20% to 40% of the code actually needs differential encoding. From the second column, we can observe that several benchmarks, like tiffmedian and ispell, have relatively small average region size in this study. In fact, we find that

Table IV. Results for Adaptive Differential Register Allocation

benchmark	#regions	ave region size	spill(%)	cost(%)	speedup
adpcm	37	21.18	3.84	3.17	1.113
blowfish	263	26.75	3.45	3.17	1.201
gsm	877	19	3.13	5.09	1.173
ispell	2492	13.94	1.9	2.68	1.144
jpeg	4646	13.44	2.02	3.64	1.148
lame	3136	15.6	1.55	2.92	1.179
patricia	20	29.01	4.36	3.77	1.149
rijndael	269	22.67	2.53	3.53	1.062
rsynth	901	18.06	2.23	6.76	1.095
sha	30	25.8	3.17	2.13	1.132
susan	429	16.68	3.82	8.25	1.1
tiffmedian	3967	12.39	2.66	3.97	1.121

more register pressure variations are present in these benchmarks, reducing the region sizes. The next column shows the percentage of spills. Compared with the spill percentage in Figure 17, spills are significantly reduced because *RegN* can be chosen up to 16 for each individual region in the adaptive scheme. This helps to supply more architected registers in high register pressure regions. On average the percentage is reduced to 2.88%.

Meanwhile, if we take a look at the cost, which includes both *set_last_reg* and *set_RegN* instructions, it is lower than the case when *RegN* is fixed to be 12 or 14. On average, the percentage of cost is 4.09% as opposed to 13.6% ($RegN = 14$) and 4.12% ($RegN = 12$) in our early results from Figure 21. The reduction of both spill and cost contributes to a higher speedup, as shown in the last column. Not surprisingly, we observe more performance improvement for most benchmarks with an average speedup of 13.5%, which is higher than the cases with fixed *RegN*s, that is, 10.2% for $RegN = 14$, 9.8% for $RegN = 12$, and 7.5% for $RegN = 10$.

The adaptive scheme is also tested on the high-performance machine model as presented in the last rows of Tables II and III. From Table II, the adaptive scheme gets higher speedup than setting *RegN* to a fixed value. The 77.48% speedup in optimized loops leads to about 0.6% more overall speedup over $RegN = 64$. However, the speedup here is not as high as that for the low-end machine model, probably because we have already excluded (turned off differential encoding for) loops with low register pressure, namely, those demanding less than 32 registers. From Table III, the number of spills is almost on par with $RegN = 64$. The small increase is probably due to the cases when spilling is less expensive than incurring high cost. The code growth is much less than setting *RegN* to a large value, for example 56 or 64, indicating the ability of the adaptive scheme to automatically scale down *RegN* to a proper value. The overall code growth is negligible (0.099%).

12.5 Evaluation for Auxiliary Optimizations

We now look at the two auxiliary optimizations introduced in Section 9. We observe that both techniques are helpful in improving the performance slightly.

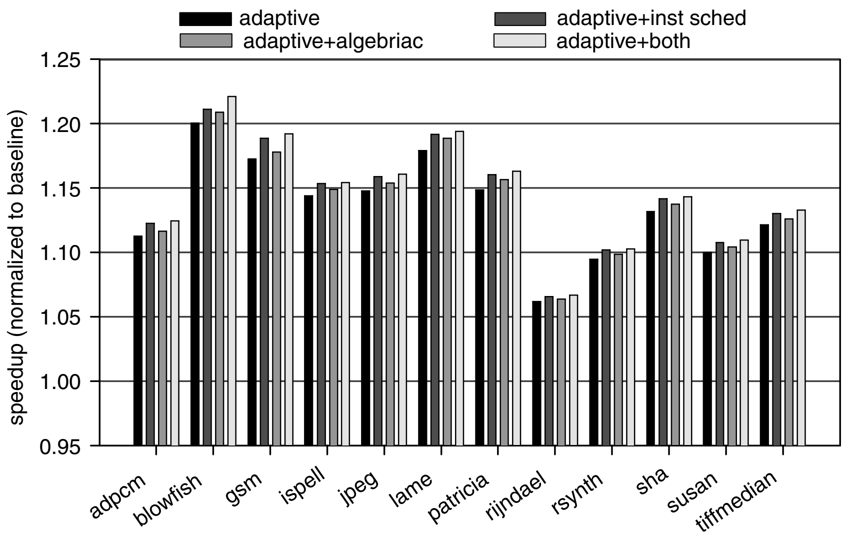


Fig. 24. Speedup comparison for auxiliary optimizations.

We cannot expect a significant speedup, since they are limited to cost reduction only. Our data is collected on the low-end machine model with differential select, since we do not intend to disrupt software pipelined code which is typically carefully scheduled. On average, both techniques add to less than 1% speedup if applied alone. Combining them yields about 1.3% speedup on top of differential select and the adaptive scheme, which achieves the largest performance enhancement, a 14.7% speedup, over the baseline.

12.6 Compilation Time

We show the results of compilation time in Table V. These results are collected on a Pentium 4 2.0 GHz machine with the GCC compiler and our enhancements. The 12 benchmarks used for the low-end system are compiled with various approaches. "remapping" induces extra compilation time but this is not very significant. Across all benchmarks, the overhead is about 7%. The percentage of increase is almost uniform, probably because the number of nodes on the adjacency graph is always equal to *RegN* for "remapping". For "select", the compilation overhead is almost negligible. The average increase is about 2%, probably because we only added a small amount of work to one of the register allocation stages.

The fifth column is for "O-spill" without "coalesce" being engaged. It must invoke the CPLEX ILP solver to find the optimal spilling during the first step, which leads to much longer compilation time than the baseline. If we look at the extra time due to "coalesce" as shown in the last column, the average slowdown is about 7%. This is due to the fact "coalesce" mainly affects the second step, while the majority of the compilation time is spent on the first.

Thus, the compilation overhead for the low-end machine is fairly low for all differential algorithms. We did not show the results for the high-end VLIW

Table V. Compilation Time (sec)

	baseline	remapping	select	O-spill	coalesce
adpcm	1.44	1.56	1.48	4.9	5.2
blowfish	1.9	2.13	1.96	13.1	13.8
gsm	4.36	4.782	4.49	103	108
ispell	16.69	18.2	17.19	210	241
jpeg	29.88	32.43	30.69	321	344
lame	6.82	7 .01	6.87	240	261
patricia	2.01	2.35	2.11	3.8	4.2
rijndael	2.65	2.78	2.69	15.6	17.8
rsynth	4.2	4.41	4.26	130	144
sha	2.84	2.96	2.88	4.7	5.1
susan	3.04	3.08	3.05	20.1	21.4
tiffmedian	9.89	10.33	10.04	430	464

machine and the adaptive scheme because the compilation slowdown is less than 2% due to the fact that differential algorithms are used only for a small portion of the code.

12.7 Hardware Cost Evaluation

In this subsection, we study the hardware cost in great detail. We first illustrate all the hardware components, then give a sample design based on our low-end model. We analyze the hardware overhead in terms of latency, energy, and area. Finally, we extend this sample design so it gives a clear picture about how much hardware cost is needed.

In Figure 25, we show all the components. Specifically, *in_reg* is the number encoded in the register field, *diff_en* a control signal indicating whether differential encoding is currently used, and finally, the *set_RegN* signals change *RegN*.

The differential enabler also receives input from *in_reg* because if it is a special-purpose register, the differential encoding is also disabled. *in_reg* can be either fed to the MUX directly when differential encoding is disabled or sent to the modulo adder for decoding. The output of the enabler controls the MUX. Meanwhile, the differential enabler also controls whether *last_reg* should be updated. If differential encoding is disabled, the output from the modulo adder is simply discarded and *last_reg* is unaffected. *in_reg* is negated on the path to the MUX to simplify the circuit (as explained in our sample design). Since the negation is not on the critical path, it does not increase the latency. The centerpiece is the modulo adder, which gets input from *in_reg*, adds to *last_reg*, and takes the modulo operation. Notice that if we do not want to change *RegN* dynamically (we can still reap most of the benefits of differential encoding), the *set_RegN* input can be avoided. Also, the modulo adder can hardwire *RegN* to make it faster. To decode multiple register fields together, we can either cascade the circuit, as in Figure 25, or do it in parallel, as mentioned in Section 2, to reduce latency. Clearly, the critical path is from the modulo adder to the MUX. It is important that these two components are designed with low latency.

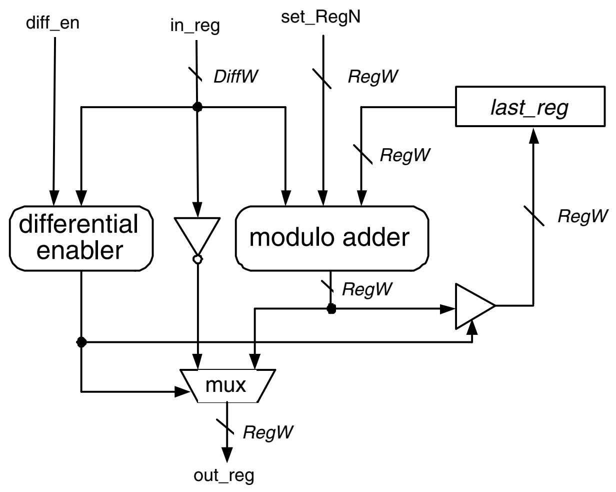


Fig. 25. Hardware circuit diagram.

Next, we show a sample design for the ARM processor. There are 16 physical registers on a typical ARM processor, so *RegW* = 4. Assume that one special register, for example, Sp, must be excluded from differential decoding. When *in_reg* = 111_b it is output directly as sp. The register field is three bits, that is, *DiffW* = 3. Figure 26(a) shows the differential enabler circuit. We only need two NAND gates. The output signal is negated, namely, zero means enabled. When *diff_en*= 1 but *in_reg* = 111_b, differential encoding is still disabled. Figure 26(b) shows the MUX circuit. The three signals on the left side are the negated *in_reg*, whereas the ones on the right are from the modulo adder. The signal from the enabler selects one of them to output. The total number of gates is 11. The modulo adder's external connection is displayed in Figure 26(c). The modulo adder can be implemented completely with two-level combinatory logic, as shown next. Notice that the four output signals are negated to simplify the MUX. Thus, we assume the hardwired *RegN* equals 16. The logic and complexity are similar for other values.

$$
\begin{aligned}
\bar{o}_0 &= \bar{d}_0\bar{r}_0 + d_1r_1\\
\bar{o}_1 &= \bar{d}_1\bar{r}_1\bar{r}_0 + \bar{d}_1\bar{d}_0\bar{r}_1 + d_1\bar{d}_0r_1 + d_1r_1\bar{r}_0 + \bar{d}_1d_0r_1r_0 + d_1d_0\bar{r}_1r_0\\
\bar{o}_2 &= \bar{d}_2\bar{d}_1\bar{d}_0\bar{r}_2 + \bar{d}_2\bar{d}_1d_0\bar{r}_2\bar{r}_1 + \bar{d}_2\bar{d}_1d_0\bar{r}_2\bar{r}_0 + \bar{d}_2\bar{d}_1d_0r_2r_1r_0 + \bar{d}_2d_1\bar{d}_0\bar{r}_2\bar{r}_1\\
&\quad + \bar{d}_2d_1\bar{d}_0r_2r_1 + \bar{d}_2d_1d_0\bar{r}_2\bar{r}_1\bar{r}_0 + \bar{d}_2d_1d_0r_2r_1 + \bar{d}_2d_1d_0r_2r_0 + d_2\bar{d}_1\bar{d}_0r_2\\
&\quad + d_2\bar{d}_1d_0r_2\bar{r}_0 + d_2\bar{d}_1d_0r_2\bar{r}_1 + d_2\bar{d}_1d_0\bar{r}_2r_1r_0 + d_2d_1\bar{d}_0r_2\bar{r}_1 + d_2d_1\bar{d}_0\bar{r}_2r_1\\
\bar{o}_3 &= \bar{d}_2\bar{d}_1\bar{d}_0\bar{r}_3 + \bar{d}_2\bar{d}_1d_0\bar{r}_3\bar{r}_2 + \bar{d}_2\bar{d}_1d_0\bar{r}_3r_2\bar{r}_1 + \bar{d}_2\bar{d}_1d_0r_2r_1 + \bar{d}_2d_1\bar{d}_0\bar{r}_3\bar{r}_2\\
&\quad + \bar{d}_2d_1\bar{d}_0\bar{r}_3r_2\bar{r}_1 + \bar{d}_2d_1\bar{d}_0r_3r_2r_1 + \bar{d}_2d_1d_0\bar{r}_3\bar{r}_2 + \bar{d}_2d_1d_0\bar{r}_3r_2\bar{r}_1\bar{r}_0\\
&\quad + \bar{d}_2d_1d_0r_3r_2\bar{r}_1\bar{r}_0 + \bar{d}_2d_1d_0r_3r_2r_1 + d_2\bar{d}_1\bar{d}_0\bar{r}_3\bar{r}_2 + d_2\bar{d}_1d_0\bar{r}_3\bar{r}_2\bar{r}_1\\
&\quad + d_2\bar{d}_1d_0\bar{r}_3\bar{r}_2r_1\bar{r}_0 + d_2\bar{d}_1d_0r_3\bar{r}_2r_1r_0 + d_2\bar{d}_1d_0r_3r_2 + d_2d_1\bar{d}_0\bar{r}_3\bar{r}_2\bar{r}_1\\
&\quad + d_2d_1\bar{d}_0r_3\bar{r}_2r_1 + d_2d_1\bar{d}_0r_3r_2
\end{aligned}
$$

Table VI. Gate Counts

	MUX	Adder	Enabler	Inverter	*last_reg* to adder	MUX to *last_reg*
# gates	11	45	2	3	4	4

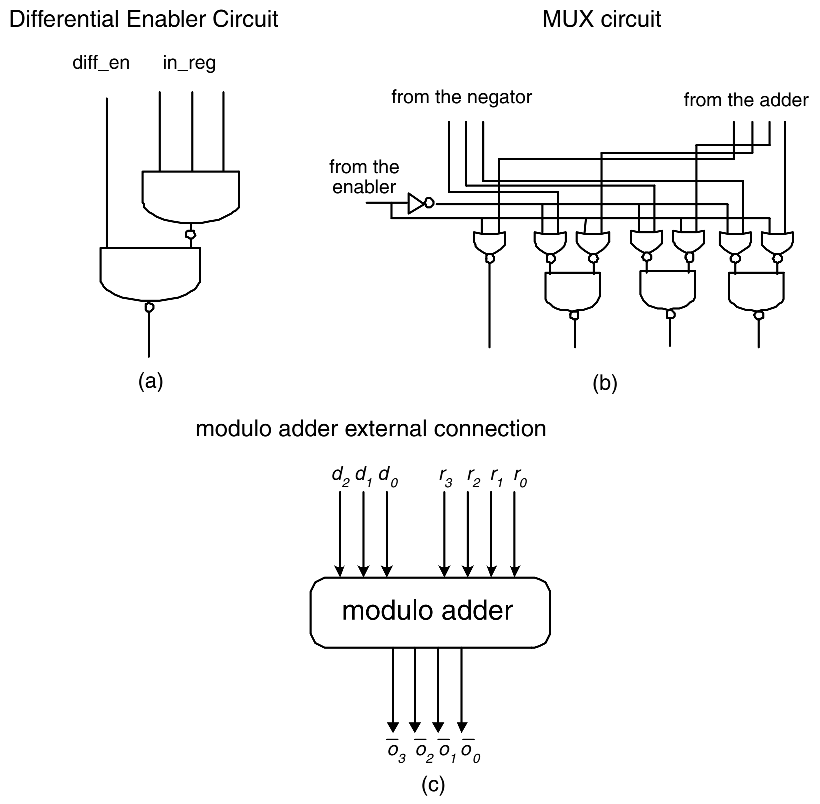


Fig. 26. A sample design for ARM.

Until this point, we have presented the complete design of the differential decoder. Therefore, we can clearly know the amount of hardware resources and the latency of all hardware components. Table VI lists the number of gates for each component. The total number of gates in the table is 80. We need to cascade three of them to decode three register fields together, thus the total number of gates for the differential decoder is up to 240. Since one gate converts to a variable number of transistors, we roughly estimate that the total number of transistors should be less than 1,500. Since even an early version of the Intel's StrongARM processor, namely, SA1110, packs 2.5 million transistors, we believe the hardware overhead is below 1,500 ÷ 2,500,000 = 0.06% for newer processors.

We also have the design for a high-end system with *RegN* = 128 physical registers. Due to space limitations, we cannot show it here. Basically, all components grow linearly with *RegN*. We can force the modulo adder to be two-level combinatory logic, but this takes more transistors (superlinear growth).

Table VII. Results for Decode Latency(ns)

	Adder	MUX	Total
0.25um(ARM)	0.124775	0.058845	0.18362
90nm(High-end)	0.09433	0.026835	0.12117

Thus, we cascade two adders so that each is the same size as in the ARM design. Although this leads to longer latency, the circuit is simpler. The number of transistors is two to three times more than that of the ARM. Considering a high-end system needs to decode on multiple paths, the total number of transistors should be less than 20 K. Now with 100–300 million transistor ICs being common, this overhead is actually below 0.02%. Based on this, we believe it is unnecessary to be concerned with power and area; in our approach already saves more power in I-cache and D-cache.

Next, we present the latency results obtained from HSPICE fact, in Table VII. The second row is for the ARM model implemented with 0.25 um technology. We calculate the latency on the critical path, namely, adder + MUX. The data listed is for decoding one register field. Supposing that we want to decode three operands, the total latency is about $0.18 \times 3 = 0.54$ ns. This could support a processor with frequency up to 2 GHz. Actually the frequency of the current ARM processor is far below 1 GHz. Assuming a 500 MHz core, the decode latency is only 1/4 of one cycle. Also, instead of cascading three decoders, we can merge them and optimize for lower latency (although it might add more transistors). In other words, the latency suffices for a real implementation on the ARM processor. There is also abundant slack for other extensions. In the third row of Table VII, we list the latency for a high-end machine with 128 physical registers, namely, $RegN = 128$ and $RegW = 7$. Here, *DiffN* is chosen to be 64. The data are collected based on 90 nm technology. The total latency is about $0.12 \times 3 = 0.36$ ns, which means it can achieve one-cycle decoding for a 3 GHz processor, even with our simple implementation. As mentioned earlier, we can use two-level combinatory logic in each decoder and optimize across decoders to achieve lower latency.

It is noteworthy that although superscalar processors might need to issue multiple instructions at once, they also have more resources that can greatly reduce the decode latency: (1) If trace cache is available, we can store differential decoded instructions there, so decoding is only needed for newly loaded instructions. (2) Since each *set_last_reg* resets the *last_reg*, making the decoding independent of previous register numbers, we can decode in parallel at each point where *last_reg* is reset. (3) Each register field must be decoded into a unique register number. If we know the register number of a particular register field, subsequent register fields can be decoded from this one. Thus, we do not need to wait for previous register fields to be decoded, and therefore, if we record some of the register numbers decoded before, we can achieve more parallelism in the decode stage.

Due to the page limitation of the article, we do not evaluate the power benefit of this scheme. Clearly, due to negligible change in code size and many less spills, both I-cache and D-cache power should be reduced.

13. RELATED WORK

Kiyohara et al. [1993] propose a scheme to make use of more physical registers by dynamically mapping architected registers to physical registers. Special instructions are added to designate the mapping and there is a mapping table managed by the hardware. Their scheme is more complicated and therefore may not be suitable for low-end machines. Compared with their work, this article proposes lightweight approaches that can be widely applicable with good performance. By making small changes in the ISA and using differential register addressing, we have shown that it is possible to overcome the addressing bottleneck in the ISA.

Zhuang et al. [2004] talk about managing extended registers, that is, registers not addressable through ISA, with hardware support. Their approach not only complicates hardware design but is less precise, since register allocation information is conveyed to the hardware through offsets of spills. The results given in Zhuang et al. [2004] only show moderate performance improvements.

Compiler-controlled memory (CCM) [Cooper and Harvey 1998] is a piece of on-chip memory for spills. The compiler can allocate spills to the CCM. However, CCM typically takes larger on-chip space than differential encoding, for example, over 1 KB, thus is slower and costs more in terms of power. By contrast, our scheme has a very small overhead. Also the CCM optimization actually can be combined with our algorithm to redirect some spills to the CCM.

George [1999] proposed a two-pass method to make better use of Intel's inaccessible (physical) registers and take advantage of the load/store forwarding. In the first pass, the register allocator allocates as if there are more registers available. Then in the second pass, it allocates to exposed registers. In this way, many spills can be converted into register moves when the hardware finds out that they are to the same address. The main difference from our work is it actually generates spills, which still causes code growth, and the hardware must be able to forward load/store values, which is not possible for most low-end systems. Moreover, once the entry in the load/store queue is removed after the memory operation finishes, the physical register is recycled for other uses. Later loads must go to the memory. By contrast, in our case, the lifetime of a value in the hidden register can be explicitly controlled by the compiler. However, we need some extra hardware to support decoding.

Other architecture solutions to extend the register file, such as the hierarchical register file [Zalamea et al. 2000b] and stack value file [Lee et al. 2001], can induce high hardware cost but do not provide a way to tackle the ISA bottleneck that restricts the number of registers seen by the compiler. Therefore, these works are orthogonal to this article.

Register windows or register stacks for example, those used on SPARC and IA-64, provide another way to expose more registers to reduce function call/context switch overhead. Differential encoding is more widely applicable because all code can be allocated with more registers.

Ravindran et al. [2003] suggest windowed register file to allow more physical registers to be utilized than does the encoding. Special window management instructions are provided to change the active window and to transfer values between windows. In contrast to the register window approach mentioned previously, their scheme is more general, therefore can be integrated with register allocation across the program. Although their work shares the same objective as ours, the approaches are quite different. Differential encoding does not require explicit instructions to change the active window, instead we can imagine there is a window being shifted implicitly every time a register field is decoded. Notice that the value in the register field is a relative number with the base specified by *last_reg*. Therefore *last_reg* can be thought of as the base of the active register window. Since *last_reg* is automatically set to the register number just decoded, the active register window is dynamically shifted upon each decoding. The advantage of our approach is that we can save those instructions to set the active window explicitly. Also, ours appears to be more smoothly integrated with a general register allocator and more widely applicable to both low-end and high-end processors.

Stack machines like Burroughs B5000, HP3000, etc., commonly have ALU operations without explicit operands, that is, the operand is always assumed to be at the top of the stack. Differential encoding also assumes a *last_reg* and the difference is always added to *last_reg*. However, the architecture and compiler changes are dramatic for stack architectures, while in this work we aim to enhance the general architecture and compiler infrastructure with the capability of differential register allocation.

Rotating register files such as those used in the Cydra 5 (and recently in IA-64) allows registers to be addressed with modulo arithmetic from a base register number that can be set/added to explicitly. Unlike in our approach, the base register cannot be changed implicitly and all these registers are compiler-addressable.

Clustered architectures like the Alpha 21264 distribute and associate register files with functional units. For Alpha 21264, allocating registers to clusters and copying values across them is managed by the hardware. For some VLIW machines, the compiler must explicitly allocate registers and copy values across clusters [Özer et al. 1998]. The goal in clustered microarchitectures is different from ours. They try to keep the on-chip structures small and fast by reducing the number of ports to register files, and by moving register files closer to the functional units. This work provides a general scheme that does not require such architecture. Also, our modification and overhead to the architecture is very small, making it applicable to a wide range of machines, especially for embedded processors with stringent encoding restrictions.

The offset assignment problem [Bartley 1992; Liao et al. 1995] studies how to utilize the postincrement/decrement addressing mode available in some processors to reduce the number of address register modification instructions. The main difference between offset assignment and this work is that we look at register accesses instead of memory accesses. Most of the research about offset assignment tackles $+1/-1$ address differences because this is most widely implemented by the hardware, while in our case the encoded differences can be wider in range. Finally, to solve the offset assignment problem, an undirected graph called the access graph is constructed and the goal is to find a path to cover the maximal edge weights. By contrast, the objective of this work as specified in Section 4 looks more complicated and the solutions are completely different, as well.

14. CONCLUSION

This article proposes differential encoding that can expose more architected registers to the compiler than the current direct encoding of the register field allows. We study three approaches to combine this with the register allocation to get better performance. Although differential encoding induces special instructions like *set_last_reg* and *set_RegN*, these instructions are much less expensive than spills. We also suggest applying it adaptively according to the register pressure and properly choosing the *RegN*/*DiffN* ratio such that more architected registers will bring about more benefits than costs. Furthermore register allocation based on differential encoding can be applied during different phases of compilation: postpass, or during select (coloring) phase or coalesce phase. Finally, the hardware modifications to implement differential encoding are minor.

We illustrate the benefits due to differential register allocation in two scenarios, namely, register allocation for a low-end system with stringent encoding limitations, and a high-end VLIW (in-order) machine where software pipelining increases register pressure and spill code.

Our experiments show that differential encoding significantly reduces the number of spills and speeds-up program execution for a low-end configuration. However, applying this technique directly to processors with ample registers could be ineffective. We then demonstrate that selectively applying it to high register pressure regions is a better way to help with the high-end processor. For the low-end configuration, it achieves over 14% speedup while keeping code size almost unaffected. It also significantly accelerates loops with high register pressure (about 80% speedup for optimized loops and 15% overall speedup) on a high-performance machine model.

For out-of-order machines, the ISA bottleneck still exists, since register renaming cannot remove dependencies that are unduly imposed by the ISA restriction. However, since out-of-order machines normally can afford more architected registers with faster code fetching/decoding, the benefits of differential RA might decrease. On the other hand, since other components are faster with out-of-order execution, the bottleneck of ISA is more detrimental to the performance.

ACKNOWLEDGMENTS

A preliminary version of this article will appear in the *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI 2005)* [Zhuang and Pande 2005]. We would like to thank anonymous PLDI referees for their comments and suggestions and also Mr. Josh Fryman from Georgia Tech for his help on the architecture side.

REFERENCES

APPEL, A. W. AND GEORGE, L. 2001. Optimal spilling for CISC machines with few registers. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 243–253.

ARM Ltd. 2007. ARM TDMI datasheet. http://www.keil.com/product/brochures/rvmdk.pdf.

Bartley, D. 1992. Optimizing stack frame access for processors with restricted addressing modes. *Softw. Pract. Exper. 22*, 2 (Feb.), 101–110.

Briggs, P., Cooper, K. D., and Torczon, L. 1994. Improvements to graph coloring register allocation. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI)*. ACM, New York.

Burger, D. and Austin, T. 1997. The SimpleScalar tool set, version 2.0 Tech. Rep. No. 1342, Computer Sciences Department, University of Wisconsin-Madison.

Chaitin, G. J., Auslander, M. A., Chandra, A. K., Cocke, J., Hopkins, M. E., and Markstein, P. W. 1981. Register allocation via coloring. *Comput. Lang. 6*, 1, 47–57.

Cooper, K. D. and Harvey, T. J. 1998. Compiler-Controlled memory. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*. 2–11.

George, L. 1999. Smlnj: Intel x86 back end compiler controlled memory. http://www.smlnj.org/compiler-notes/k32.ps.

George, L. and Appel, A. W. 1996. Iterated register coalescing. *ACM Trans. Program. Lang. Syst. 18*, 3, 300–324.

Guthaus, M. R., Ringenberg, J. S., Ernst, D., Austin, T. M., Mudge, T., and Brown, R. B. 2001. Mibench: A free, commercially representative embedded benchmark suite. In *IEEE 4th Annual Workshop on Workload Characterization*. IEEE.

Intel Inc. 1998. *SA-110 Microprocessor Technical Reference Manual*. Intel, Santa Clara, CA.

Kiyohara, T., Mahlke, S., Chen, W., Bringmann, R., Hank, R., Anik, S., and Hwu, W.-M. 1993. Register connection: A new approach to adding registers into instruction set architectures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*. 247–256.

Krishnaswamy, A. and Gupta, R. 2002. Profile guided selection of ARM and thumb instructions. In *ACM SIGPLAN Joint Conference on Languages Compilers and Tools for Embedded Systems (LCTES)*. ACM, New York.

Lam, M. S.-L. 1987. *A Systolic Array Optimizing Compiler*. Carnegie Mellon Pittsburgh, PA.

Lee, H.-H. S., Smelyanskiy, M., Tyson, G. S., and Newburn, C. J. 2001. Stack value file: Custom microarchitecture for the stack. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture (HPCA)*.

Liao, S., Devadas, S., Keutzer, K., Tjiang, S., and Wang, A. 1995. Storage assignment to decrease code size . In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI)*. ACM, New York.

Llosa, J., Valero, M., and Ayguadé, E. 1996. Heuristics for register-constrained software pipelining. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*. 250–261.

MIPS Technologies. 2001. *MIPS32 Architecture for Programmers, volume IV-a: The MIPS16 Application Specific Extension to the MIPS32 Architecture*. MIPS Technologies.

Motorola Inc. 2000. *Motorola DSP56300 Family Manual*, revision 3.0. Motorola, Phoenix, AZ.

Özer, E., Banerjia, S., and Conte, T. M. 1998. Unified assign and schedule: A new approach to scheduling for clustered register file microarchitectures. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*. 308–315.

Rau, B. R., Lee, M., Tirumalai, P. P., and Schlansker, M. S. 1992. Register allocation for software pipelined loops. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 283–299.

Ravindran, R. A., Senger, R. M., Marsman, E. D., Dasika, G. S., Guthaus, M. R., Mahlke, S. A., and Brown, R. B. 2003. Increasing the number of effective registers in a low-power processor using a windowed register file. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. 125–136.

Ruttenberg, J., Gao, G. R., Stoutchinin, A., and Lichtenstein, W. 1996. Software pipelining showdown: Optimal vs. heuristic methods in a production compiler. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 1–11.

Segars, S. 2001. Low power design techniques for micro-processors. In *Tutorial on IEEE International Solid-State Circuits Conference (ISSCC)*.

Wang, J., Krall, A., Ertl, M. A., and Eisenbeis, C. 1994. Software pipelining with register allocation and spilling. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*. 95–99.

Zalamea, J., Llosa, J., Ayguadé, E., and Valero, M. 2000a. Improved spill code generation for software pipelined loops. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 134–144.

Zalamea, J., Llosa, J., Ayguadé, E., and Valero, M. 2000b. Two-Level hierarchical register file organization for vliw processors. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*. 137–146.

Zhuang, X. and Pande, S. 2005. Differential register allocation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. to appear.

Zhuang, X., Zhang, T., and Pande, S. 2004. Hardware-Managed register allocation for embedded processors. In *Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools*. 192–201.

Received March 2005; revised August 2005; accepted November 2005