Lifetime Holes Aware Register Allocation for Clustered VLIW Processors

Xuemeng Zhang*, Hui Wu† and Haiyan Sun* and Jingling Xue†
*School of Computer, National University of Defense Technology, Changsha, China
Email: {xuemengz,shy}@hotmail.com
†School of Computer Science and Engineering, The University of New South Wales, Sydney, Australia
Email: {huiw,jingling}@cse.unsw.edu.au

***Abstract*—This paper presents an on-the-fly register allocator which dynamically detects and utilises lifetime holes for clustered VLIW processors. A lifetime hole is an interval in which a variable does not contain a valid value. A register holding a lifetime hole can be allocated to another variable whose live range fits in the lifetime hole, leading to more efficient utilisation of registers. We propose efficient techniques for dynamically utilising lifetime holes and incorporate these techniques into our on-the-fly register allocator. We have simulated our register allocator and a linear scan register allocator without considering lifetime holes by using the MediaBench II benchmark suite. Our simulation results show that our register allocator reduces the number of spills by 12.5%, 11.7%, 12.7%, for three different processor models, respectively.**

***Index Terms*—register allocation; lifetime hole; live range; clustered VLIW processor; inter-cluster communication**

I. Introduction

Register allocation is an important component of an optimising compiler for VLIW (Very Long Instruction Word) processors. Traditionally, register allocation is performed separately from instruction scheduling, causing the phase ordering problem. For clustered VLIW processors, since clusters are connected by an inter-cluster communication network, performing register allocation before instruction scheduling may cause unnecessary inter-cluster communications. To alleviate the phase ordering problem, it is desirable to integrate register allocation and instruction scheduling into a single phase.

Clustering is a well-known technique for improving the scalability and energy consumption of VLIW processors. However, clustered VLIW processors make register allocation more challenging. Firstly, new live intervals may be created dynamically when values are transferred to different clusters. Secondly, the exact live range of a variable depends on when the operations of its first definition and last use are scheduled, which cannot be determined by traditional liveness analysis for static code. In addition, bad cluster assignment may cause unnecessary inter-cluster communications, thus increasing the schedule length of a basic block.

A lifetime hole of a variable is an interval during which the variable does not hold a valid value. If a register R_0 is assigned to a variable v and v has a lifetime hole, R_0 can still be allocated to another variable w, provided that w's live range fits in the lifetime hole of v. Registers can be reused during lifetime holes to improve the utilisation of registers effectively.

978-3-9815370-2-4/DATE14/@2014EDAA

We propose an on-the-fly register allocator which detects and utilizes lifetime holes for clustered VLIW processors. To our knowledge, our register allocator is the first one considering lifetime holes for clustered VLIW processors. Our register allocator is integrated with a priority based instruction scheduler [1], [2]. When the instruction scheduler schedules an operation of a basic block, the instruction scheduler calls the register allocator to assign physical registers to the virtual registers of the operation. We have simulated our register allocator and compared it to a linear scan register allocator without considering lifetime holes proposed in [3] using the MediaBench II benchmark suite [4] based on three different processor models. On average, our register allocator reduces the number of spills by 12.5%, 11.7%, 12.7% compared to the linear scan register allocator for the three processor models, respectively.

II. Related Work

Two common approaches used in modern compilers are graph colouring and linear scan. Graph colouring is suitable when compilation time is not a major concern, while linear scan is faster and therefore frequently used for just-in-time compilers. Chaitin et al. [5] introduce the first a graph colouring register allocator based on an interference graph G. The problem of allocating k registers to the variables is to look for a k-colouring of G such that adjacent nodes always have distinct colours. Matula et al. [6] and Briggs et al. [7] improve on Chaitin's work. Chow et al. [8] present a scheme to colour the graph using a priority ordering. Poletto et al. [3] propose a linear scan register allocation algorithm, which allocates registers to variables in a single linear-time scan of the variables' live ranges. Traub et al. [9] propose a more complex linear scan algorithm, which utilises lifetime holes by assigning two variables to the same register if the live range of one register is entirely contained in a lifetime hole of the other. Sarkar and Barik [10] propose two extended linear scan algorithms that retain the compile-time efficiency of the previous linear scan algorithms, while delivering performance that can match or surpass that of graph colouring. Wimmer and Mössenböck [11] present an optimised implementation of the linear scan algorithm for the Java HotSpot^{TM} client compiler.

III. Lifetime Holes

Given a program P, let B_0, B_1, $\cdots$, B_{n-1} be a list B of all the basic blocks of the program, sorted in reverse postorder [12] which can reduce the lifetime holes [9] and improve the quality of register allocation. Define the rank of each basic block B_i, denoted $rank(B_i)$, as its position in the list, i.e., $rank(B_i) = i$. Let $\alpha(v)$ be the start time of an operation v. Define an exact lifetime hole $h(x)$ of a variable x as a two dimensional interval: $h(x)$ = $[rank_i(x)$, $rank_j(x)]$ $(i \leq j)$, where $rank_i(x)$ = $(rank(B_i)$, $\alpha(v))$: x is live at operation v and not live after operation v in the basic block B_i, $rank_j(x)$ = $(rank(B_j)$, $\alpha(w)$: x is live after operation w in the basic block $B_j)$, and x is not live between v and w. Initially, $h(x)$ is set to $[(rank(B_i), 0)$, $(rank(B_j), -1)]$, or $[(rank(B_i), -1)$, $(rank(B_j), -1)]$, or $[(rank(B_i), -1)$, $(rank(B_j), \epsilon)]$, where -1 indicates an unknown cycle, 0 indicates the start of a basic block, and ϵ indicates the end of a basic block. $h(x)$ will be updated when the corresponding operations are scheduled.

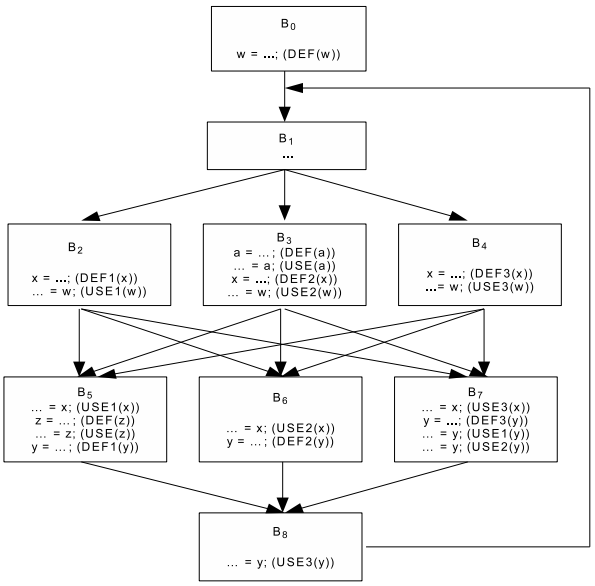

Fig. 1: An example CFG

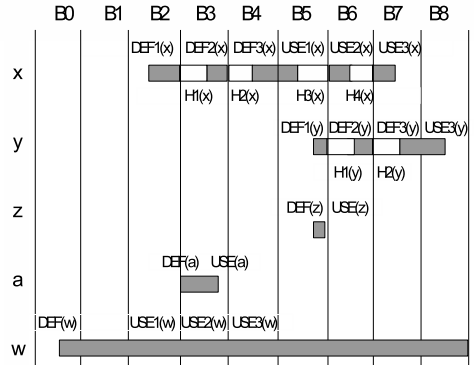

Fig. 2: Reverse postorder of the example CFG in Fig. 1 and the lifetime holes

To ensure that the value in the register allocated to a live range is valid, we utilise a lifetime hole of the live range only after its start operation is scheduled. In addition, we reuse the register with the lifetime hole only for another variable whose lifetime ends earlier than the end of the lifetime hole. Consider Fig. 2. Initially $H3(x)$ is $[(5, -1)$, $(5, \epsilon)]$. Assume that $USE1(x)$ is scheduled at cycle 20, $H3(x)$ = $[(5, 20)$, $(5, \epsilon)]$. Any operation whose live range is between $(5, 20)$ and $(5, \epsilon)$ can reuse x's register.

IV. Lifetime Hole Detection

Algorithm 1 $HolesDetection(B, in, out, H)$

Require:
A list B of all the basic blocks of the CFG sorted in reverse postorder
A list in of the variables live-in at each basic block
A list out of the variables live-out at each basic block
Ensure:
The lifetime holes $H(x)$ of each variable x

```
1:  Let def_l be the set of variables defined by each basic block B_l in B
2:  Let use_l be the set of variables used in each basic block B_l in B
3:  Let use = ∪use_l, l ∈ (0, |B| − 1)
4:  Let α(v) be the start time of operation v
5:  for each variable x ∈ use do
6:      Let [(rank_min(x), −1), (rank_max(x), −1)] be x's live range
7:      i = rank_min(x)
8:      while i ≤ rank_max(x) do
9:          i++;
10:         if x ∉ in_i then
11:             bstart = i;
12:             i++;
13:             while i ≤ rank_max(x) ∧ x ∉ in_i do
14:                 i++;
15:             end while
16:             bend = i − 1;
17:             Add lifetime hole [(bstart, 0), (bend, ϵ)] into H(x)
18:         end if
19:         i++;
20:     end while
21:     for each basic block B_l in the list B do
22:         if x ∈ use_l then
23:             if x ∉ out_l ∧ l < rank_max(x) then
24:                 Let w be the last use of x in B_l
25:                 Add h(x) = [(l, α(w)), (l, ϵ)] into H(x)
26:             end if
27:         else if x ∈ def_l ∧ l > rank_min(x) then
28:             Let v be the first definition of x in B_l
29:             Add h(x) = [(l, 0), (l, α(v))] into H(x)
30:         end if
31:     end for
32:     Merge the adjacent lifetime holes into a single lifetime hole
33: end for
```

As our register allocator is performed in the same phase as instruction scheduling, lifetime holes cannot be known and utilised based on static code. Our register allocator detects lifetime holes dynamically during instruction scheduling. The algorithm for detecting lifetime holes is shown in Alg. 1. This algorithm has three parts. The first part (line 5 to line 20) detects lifetime holes which are longer than one basic block. The second part (line 21 to line 29) detects lifetime holes that are within one basic block. These lifetime holes are created by multiple last uses or multiple first definitions in the different branches. The last part (line 32) merges all the adjacent lifetime holes into a single lifetime hole.

Assume that all the live ranges are known by traditional liveness analysis [13] and it takes $O(1)$ time to check each

of the conditions: $x \in use_l$, $x \notin out_l$, and $x \in def_l$. Each of the three parts takes $O(|B| * n)$ time, where $|B|$ is the number of basic blocks of the CFG and n is the number of variables in the CFG. Therefore, the time complexity of the algorithm is $O(|B| * n)$. After lifetime holes are detected, the exact live range of a variable can be computed by removing all the lifetime holes from the original live range.

Given a variable x, let $rank_{\min}(x)$=min$\{rank(B_i)$: B_i contains a definition of $x\}$, $rank_{\max}(x)$=max$\{rank(B_i)$:B_i contains a use of $x\}$, $\beta(x)$ be the start time of the operation of x's first definition in B, and $\gamma(x)$ be the start time of the operations of x's last use in B. If the operation of x's first definition in B is not scheduled yet, $\beta(x)$ is set to -1. Similarly, if the operation of x's last use in B is not scheduled yet, $\gamma(x)$ is set to -1. A live range of a variable x is a two dimensional interval defined as $\tau(x)$=$[start(x), end(x)]$, where $start(x)$=$(rank_{min}(x), \beta(x))$. For the definition of $end(x)$, we need to distinguish the following two cases.

1) The live range of x is not in a loop, or the live range of x is in a loop, but does not contain the entire loop. We define $end(x)$ as follows: $end(x)$=$(rank_{max}(x)$, $\gamma(x))$.
2) The live range of x is in a loop, and contains the entire loop. Let $loop_{max}$=max$\{rank(B_i)$:B_i is inside the loop$\}$. In this case, $end(x)$=$(loop_{max}, \epsilon)$, where ϵ indicates the end of a basic block.

Initially, the live range $\tau(x)$ of a variable x is set to $[(rank_{min}(x), -1), (rank_{max}(x), -1)]$ or $[(rank_{min}(x), -1), (loop_{max}, \epsilon)]$. It will be updated when the corresponding operations are scheduled. Consider w and x in Fig. 1, initially $\tau(w)$ = $[(0, -1), (8, \epsilon)]$ and $\tau(x)$ = $[(2, -1), (7, -1)]$. Assume that $DEF(w)$ is scheduled at cycle 1 in the schedule of B_0, the operation of x's first definition is scheduled at cycle 5 in the schedule of B_2, and the operations of x's last use is scheduled at cycle 10 in the schedule of $B7$. We have $\tau(w)$ = $[(0, 1), (8, \epsilon)]$ and $\tau(x)$ = $[(2, 5), (7, 10)]$.

As lifetime holes are detected dynamically, our on-the-fly register allocator utilises lifetime holes in a conservative way according to the dynamical scheduling, which guarantees that each register always holds a valid value. Given a lifetime hole $h(x) = [(i, t_1), (j, t_2)]$ $(i < j)$, of a variable x, for any variable y with a live range $\tau(y) = [(m, t_3), (n, t_4)]$ $(m \leq n)$, $h(x)$ can be reused by y if both of the following two constraints are satisfied:

1) y's live range starts later than the starting point of $h(x)$,
   a) $i < m$ holds, or
   b) $i = m$ and $t_3 > t_1$ hold.
2) y's live range ends earlier than the end point of $h(x)$,
   a) $n < j$ holds. Or
   b) $n = j$ and $t_2 = \epsilon$ hold. In this case, $h(x)$ ends in the end of the basic block and $\tau(y)$ ends at an operation within this basic block. Or
   c) $n = j$ holds, and x is data dependent on y in B_j. In this case, $\tau(y)$ definitely ends earlier than $h(x)$.

V. Register Allocation with Lifetime Holes

Alg. 2 shows our on-the-fly register allocator considering lifetime holes. When scheduling each ready operation v_i, if v_i needs a value produced by another operation v_j, and the value has been spilled or v_j is not on the same cluster as v_i, we allocate a register to store the value by finding a lifetime hole long enough, as shown from line 1 to line 26. After that, from line 27 to line 33, we allocate a register to the destination virtual register of v_i if necessary, by finding a lifetime hole long enough for the entire live range of the destination virtual register. Otherwise, a register will be spilled for v_i. As free registers can be viewed as holding a very long lifetime hole until it is allocated, allocating registers to variables can be viewed as allocating lifetime holes to variables. Therefore, our register allocator is also called lifetime hole allocator.

Our lifetime hole allocator allocates registers efficiently according to the following two rules. Firstly, our lifetime hole allocation strategy not only allocates free registers to variables, but also allocates registers holding lifetime holes long enough to variables. Either of the following two cases show that a register can be allocated to variable:

1) The bit in TA that is corresponding to the register is -1, which means the register is free.
2) Or the bit in TA which is corresponding to the register is not -1, and the bit in TH which is related to the register is 1, which means the register is holding a lifetime hole.

When a lifetime hole is utilised, the corresponding bit in TH is updated to be 0. Secondly, selecting a register equals to selecting a lifetime hole long enough for a variable to fit in. Our lifetime hole allocation strategy chooses the lifetime hole which ends the earliest among all the lifetime holes that end later than the variable's live range.

VI. Simulation Results

To evaluate the performance of our register allocator, we have integrated it into a priority based instruction scheduler [1], [2]. When scheduling an operation of each basic block, the scheduler calls register allocator to assign physical registers to virtual registers of the operation. We have also integrated a linear scan register allocator without considering lifetime holes proposed in [3] into the scheduler. To compare both register allocators, we use the MediaBench II benchmark suite [4]. The hardware platform for the simulation is an Intel(R) Core(TM)2 Quad CPU Q9400 with a clock frequency of 2.66GHz, 3.5 GB memory, and 3MB cache. We use three different clustered VLIW processor models, Processor (a), (b), and (c), with 4, 3, and 2 clusters, respectively. All the three processor models are based on the TMS320C6X processor [14]. All the clusters of each processor model are identical. Each cluster has a L, a S, a D, and a M unit. The instruction set is the same as that of the TMS320C6X processor. The inter-cluster communication latency is 3 cycles. The latencies of $load$, $multiply$ and $branch$ operations are 4, 1 and 5 cycles, respectively. The latencies of other operations are 0 cycles. In addition, the register file of each cluster has 16 registers.

Algorithm 2 $HoleAllocator(M, V_i, TA, TS, TR, TH)$

Require:
A clustered VLIW processor M with $|R|$ registers on each cluster
A ready operation V_i to be scheduled on cluster C_m
A register availability table TA
A spill table TS
A register allocation table TR
A lifetime hole table TH of all the registers on M
Ensure:
A updated register availability table TA
A updated spill table TS
A updated register allocation table TR
A updated lifetime hole table TH of M

```
1: if V_i needs the results of a set P of other operations then
2:     for j = 0 to |P| − 1 do
3:         if P_j is in the memory then
4:             if there is a lifetime hole for P_j on C_m then
5:                 Allocate the register holding the lifetime hole to P_j, and
                   update TR, TA, TH
6:             else
7:                 Choose a register to spill for P_j, and update TS
8:                 Add a store operation on C_m
9:                 Allocate the register to P_j, and update TR, TA and TH
10:            end if
11:            Add a load operation to reload P_j on C_m
12:        end if
13:        if P[j] is not on C_m then
14:            if there is a lifetime hole for P_j on C_m then
15:                Allocate the register holding the lifetime hole to P_j, and
                   update TR, TA, TH
16:            else
17:                Choose a register to spill for P_j, and update TS
18:                Add a store operation on C_m
19:                Allocate the register to P_j, and update TR, TA and TH
20:            end if
21:        end if
22:        if P_j's live range ends at V_i then
23:            Free P_j's register and update TA
24:        end if
25:    end for
26: end if
27: if there is a lifetime hole for V_i then
28:    Allocate the register holding the lifetime hole to V_i, and update TR,
       TA and TH
29: else
30:    Choose a register to spill for V_i and update TS
31:    Add a store operation on C_m
32:    Allocate the register to V_i, and update TR, TA and TH
33: end if
```

The total number of spills in each benchmark is shown in Fig. 3. The simulation results shown in Fig. 3(i) are obtained by using Processor (i). In each figure, a vertical axis represents the total number of spills of a benchmark on a specific clustered VLIW processor. The column $Hole$ indicates our register allocator, and the other column $NoHole$ indicates the linear scan register allocator without considering lifetime holes. For all the benchmarks, our register allocator has fewer spills than the linear scan register allocator. On average, our register allocator reduces the number of spills by 12.5%, 11.7%, 12.7% for the three processor models, respectively. The key reason is that our register allocator detects and utilises lifetime holes, reducing the register pressure.

VII. Conclusion

We have proposed a lifetime holes aware, on-the-fly register allocator for clustered VLIW processors. Our register allocator detects and utilises lifetime holes dynamically, leading to better register utilisation. When allocating a physical register to a virtual register, the lifetime hole allocator selects the physical register holding the lifetime hole which ends the earliest among all the registers holding lifetime holes that are long enough. The simulation results show that our register allocator effectively reduces the number of spills for a set of benchmarks from the MediaBench II benchmark suite.

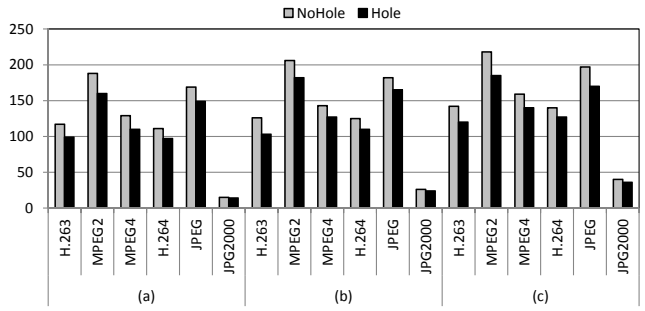

Fig. 3: Total number of spills for each benchmark

VIII. Acknowledgement

This work is sponsored by the National Natural Science Foundation of China under Grant No.61303072.

References

[1] X. Zhang, H. Wu, and J. Xue, "An Efficient Heuristic for Instruction Scheduling on Clustered VLIW Processors," in *Proceedings of 2011 International Conference on Compilers, Architectures and Synthesis of Embedded Systems*, 2011.

[2] X. Zhang, H. Wu, and J. Xue, "Instruction Scheduling with K-successor Tree for Clustered VLIW Processors," *The Journal of Design Automation for Embedded Systems*, 2013.

[3] M. Poletto and V. Sarkar, "Linear Scan Register Allocation," in *ACM Transactions on Programming Languages and Systems*, vol. 21, 1999.

[4] mediabench, "Mediabench II Benchmark," *http://euler.slu.edu/ fritts/mediabench/*.

[5] G. J. Chaitin, "Register Allocation and Spilling via Graph Coloring," in *Proceedings of the ACM SIGPLAN 82 Symposium on Compiler Construction*, S. Notices, Ed., vol. 17(6), 1982, pp. 98–105.

[6] D. W. Matula and L. L. Beck, "Smallest-last Ordering and Clustering and Graph Coloring Algorithms," in *Journal of the Association for Computing Machinery*, vol. 30, 1983, pp. 417–427.

[7] P. Briggs, K. D. Cooper, and L. Torczon, "Improvements to Graph Coloring Register Allocation," in *ACM Transactions on Programming Languages and Systems*, vol. 16, 1994, pp. 428–455.

[8] F. Chow and J. Hennessy., "The Priority-based Coloring Approach to Register Allocation," in *ACM Transactions on Programming Languages and Systems*, vol. 12, 1990, pp. 501–536.

[9] O. Traub, G. Holloway, and M. D. Smith, "Quality and Speed in Linear-scan Register Allocation," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1998, pp. 142–151.

[10] V. Sarkar and R. Barik, "Extended Linear Scan: An Alternate Foundation for Global Register Allocation," in *Proceedings of the International Conference on Compiler Construction*, 2007, pp. 141–155.

[11] C. Wimmer and H. Mössenböck, "Optimized Interval Splitting in a Linear Scan Register Allocator," in *Proceedings of the ACM/USENIX International Conference on Virtual Execution Environments*, 2005.

[12] J. B. N. Kam and J. D. Ullman, "Global Data Flow Analysis and Iterative Algorithms," *Journal of the Association for Computing Machinery*, vol. 23, no. 1, pp. 158–171, 1976.

[13] M. S. Hecht and J. D. Ullman, "A Simple Algorithm for Global Data Flow Analysis Problems," *SIAM Journal on Computing*, vol. 4, no. 4, pp. 519–532, 1975.

[14] "TI TMS320C64xx DSPs," *http://www.ti.com*.