A Dynamic Data Dependence Analysis Approach for Software Pipelining

Lin Qiao, Weitong Huang, Zhizhong Tang

Department of Computer Science and Technology, Tsinghua University,
Beijing, 100084, PR China
{qiaolin, hwt}@cic.tsinghua.edu.cn; tzz-dcs@tsinghua.edu.cn

Abstract. This paper presents a run-time pointer aliasing disambiguation method for software pipelining techniques. By combining hardware with software, the method is better than run-time checking method or run-time compensation method, which is capable of dealing with irreversible code, and has limited compensation code space without serious rerollability problem. The new method solves pointer aliasing problem efficiently and makes it possible to obtain potential instruction-level parallel speedup. In this paper instruction-level parallel speedups of the new method are analyzed in detail. Three theoretical speedups, i.e., *general speedup*, *probabilistic speedup* and *mean speedup with probability*, are given, which will be helpful for studying and evaluating instruction-level parallelism of the new method.

1 Introduction

To exploit instruction-level parallelism (ILP), compilers for a very-long instruction word (VLIW) machine often employ static code scheduling and software pipelining [1] [2] [3] [4]. It is, however, restricted by ambiguous dependencies between memory fetches. Even though great progress has been made in the analysis of static aliases among arrays, analysis of pointer aliasing is a formidable task for most compilers. In order to solve this key problem to achieving the potential speedup in instruction-level parallel processing, two types of run-time disambiguation (RTD) methods, i.e., run-time checking and run-time compensation, have been presented in [5].

When applying both of the run-time disambiguation methods to software pipelining, however, the run-time compensation approach allows speculative memory fetch but is suitable only for reversed code, while the run-time checking approach can be used for any code but has serious rerollability problem. Moreover, both of the run-time disambiguation methods have code space problem. In particular, when applying run-time disambiguation to global software pipelining the space of compensation code could be tremendous.

Followed Su and his co-operators [6], the paper presents a new hardware/software combined method. The basic ideas are as follows. First, during run time, let the function units execute NOP operations instead of using compensation code to implement the postponement of the incorrect memory load operation and its successive operations. Second, to guarantee the consistency of the execution sequence

of all postponed operations, the order of function units that execute NOPs and the number of NOPs must be determined during compiler time.

This paper is organized as follows. Section 2 discusses the hardware support for the RTPAD method before how to use the RTPAD method is discussed by a sample example in detail in Section 3. Section 4 presents three theoretical parallel speedups and analyzes the example. Section 5 gives some experimental results while Section 6 draws conclusions.

2 Hardware Architecture

Fig.1 illustrates a hypothetical VLIW architecture that has ten function units: two ALU, two multipliers (MUL), two memory ports (MEM), and four branch-and-loop-control units (BRLC). In addition, the hardware support of the RTPAD method includes an instruction buffer (IB) storing postponed operations, a multiplexer set (MUX) selecting operations from regular instruction memory or from instruction buffer, an RTPAD control instruction buffer, and a read register called RTPAD WORD. This VLIW processor is capable of starting four integer operations, two memory operations, and four branch operations every cycle.

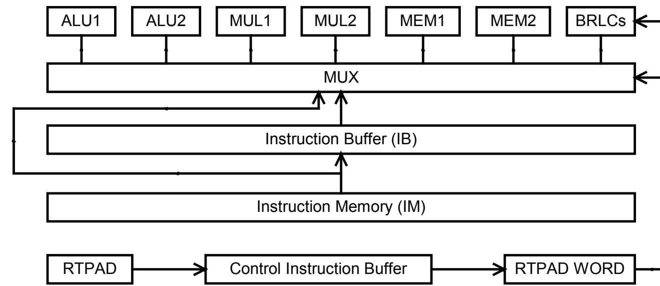

Fig. 1. Hardware support for the RTPAD method

3 Using the RTPAD Method

The RTPAD method has been used for software pipelining of non-loop programs in [7] [8] [9]. This paper extends the work by using it for software pipelining of loop programs.

Table 1 and Table 2 illustrates how to use the RTPAD method for software pipelining algorithms. Fig. 2 shows the original code and Fig. 3 shows the modified code into which RTPAD operations are inserted.

Because the software pipelining algorithm overlaps several iterations, some RTPAD operations are inserted before the ambiguous load operation as shown in Fig. 3. Table 1 shows the normal execution sequence of the result of software pipelining when no address conflict is detected, where $op_i^{(j)}$ denotes operation op_i belongs to the j-th iteration of the loop.

Table 1. The result of software pipelining without address conflicts

CLK	ALU1	ALU2	MUL1	MUL2	MEM1	MEM2	BRLC0	BRLC1	BRLC2
1			$op_1^{(1)}$						
2			$op_1^{(2)}$		$op_5^{(1)}$				RTPAD($op_5^{(2)}$, $op_8^{(1)}$)
3	$op_6^{(1)}$		$op_1^{(3)}$		$op_5^{(2)}$			RTPAD($op_5^{(3)}$, $op_8^{(1)}$)	RTPAD($op_5^{(3)}$, $op_8^{(2)}$)
4	$op_6^{(2)}$	$op_7^{(1)}$	$op_1^{(4)}$		$op_5^{(3)}$		RTPAD($op_5^{(4)}$, $op_8^{(1)}$)	RTPAD($op_5^{(4)}$, $op_8^{(2)}$)	RTPAD($op_5^{(4)}$, $op_8^{(3)}$)
5	$op_6^{(3)}$	$op_7^{(2)}$	$op_1^{(5)}$		$op_5^{(4)}$	$op_8^{(1)}$	RTPAD($op_5^{(5)}$, $op_8^{(2)}$)	RTPAD($op_5^{(5)}$, $op_8^{(3)}$)	RTPAD($op_5^{(5)}$, $op_8^{(4)}$)
6	$op_6^{(4)}$	$op_7^{(3)}$	$op_1^{(6)}$	$op_9^{(1)}$	$op_5^{(5)}$	$op_8^{(2)}$	RTPAD($op_5^{(6)}$, $op_8^{(3)}$)	RTPAD($op_5^{(6)}$, $op_8^{(4)}$)	RTPAD($op_5^{(6)}$, $op_8^{(5)}$)
7	$op_6^{(5)}$	$op_7^{(4)}$	$op_1^{(7)}$	$op_9^{(2)}$	$op_5^{(6)}$	$op_8^{(3)}$	RTPAD($op_5^{(7)}$, $op_8^{(4)}$)	RTPAD($op_5^{(7)}$, $op_8^{(5)}$)	RTPAD($op_5^{(7)}$, $op_8^{(6)}$)
8	$op_6^{(6)}$	$op_7^{(5)}$	$op_1^{(8)}$	$op_9^{(3)}$	$op_5^{(7)}$	$op_8^{(4)}$	RTPAD($op_5^{(8)}$, $op_8^{(5)}$)	RTPAD($op_5^{(8)}$, $op_8^{(6)}$)	RTPAD($op_5^{(8)}$, $op_8^{(7)}$)
9	$op_6^{(7)}$	$op_7^{(6)}$	$op_1^{(9)}$	$op_9^{(4)}$	$op_5^{(8)}$	$op_8^{(5)}$	RTPAD($op_5^{(9)}$, $op_8^{(6)}$)	RTPAD($op_5^{(9)}$, $op_8^{(7)}$)	RTPAD($op_5^{(9)}$, $op_8^{(8)}$)

Table 2. An address conflict between $op_5^{(6)}$ and $op_8^{(4)}$ is detected

CLK	ALU1	ALU2	MUL1	MUL2	MEM1	MEM2	BRLC0	BRLC1	BRLC2
6	$op_6^{(4)}$	$op_7^{(3)}$	$op_1^{(6)}$	$op_9^{(1)}$	$op_5^{(5)}$	$op_8^{(2)}$	RTPAD($op_5^{(6)}$, $op_8^{(3)}$)	RTPAD($op_5^{(6)}$, $op_8^{(4)}$)	RTPAD($op_5^{(6)}$, $op_8^{(5)}$)
7	$op_6^{(5)}$	$op_7^{(4)}$	NOP	$op_9^{(2)}$	NOP	$op_8^{(3)}$	RTPAD($op_5^{(7)}$, $op_8^{(4)}$)	RTPAD($op_5^{(7)}$, $op_8^{(5)}$)	RTPAD($op_5^{(7)}$, $op_8^{(6)}$)
8	NOP	$op_7^{(5)}$	NOP	$op_9^{(3)}$	NOP	$Op_8^{(4)}$	RTPAD($op_5^{(8)}$, $op_8^{(5)}$)	RTPAD($op_5^{(8)}$, $op_8^{(6)}$)	RTPAD($op_5^{(8)}$, $op_8^{(7)}$)
9	NOP	NOP	$op_1^{(7)}$	NOP	$op_5^{(6)}$	NOP	NOP	NOP	NOP
10	$op_6^{(6)}$	NOP	$op_1^{(8)}$	NOP	$op_5^{(7)}$	NOP	NOP	NOP	NOP
11	$op_6^{(7)}$	$op_7^{(6)}$	$op_1^{(9)}$	$op_9^{(4)}$	$op_5^{(8)}$	$op_8^{(5)}$	RTPAD($op_5^{(9)}$, $op_8^{(6)}$)	RTPAD($op_5^{(9)}$, $op_8^{(7)}$)	RTPAD($op_5^{(9)}$, $op_8^{(8)}$)

```
for( i=0; i<n; i++ )
{
 R2 = 2 * R1
 R1 = M(P)
 R4 = R2 – R1
 R4 = R4 + R3
 M(Q) = R6
 R7 = R4 * R5
}
```

Fig. 2. The original code

```
for( i=0; i<n; i++ )
{
op1: R2 = 2 * R1
op2: RTPAD
op3: RTPAD
op4: RTPAD
op5: R1 = M(P)
op6: R4 = R2 – R1
op7: R4 = R4 + R3
op8: M(Q) = R6
op9: R7 = R4 * R5
}
```

Fig. 3. After RTPAD inserted

The prologue stage of the loop is from cycle 1 to cycle 5, and the pipelining stage of the loop begins from cycle 6. In Table 1, each VLIW instruction executes 6 operations belonging to adjoining iterations, namely, it takes a VLIW CPU one cycle to complete an iteration of the loop. Assume that l be the loop length and n be the loop counter. If $n >> l$, the corresponding parallel speedup is l approximately.

Three RTPAD operations are inserted to determine whether memory address conflict between the ambiguous load operation of the iteration and store operations of previous three iterations, respectively. As Table 2 shows, all operations at cycle 11 are the same as original run-time VLIW code at cycle 9, which means that all operations within cycle 7 and cycle 8 in Table 1 are performed within cycle 7 to cycle 10 in Table 2. All data dependencies of these operations are guaranteed by the order of inserted NOP operations. The RTPAD method totally needs two extra cycles to complete compensation NOP operations, which is equal to the compensation code measure, when the address conflict is detected.

It takes a sequential CPU 6n cycles to execute the original code as shown in Fig. 2. When the RTPAD method is used, it takes a VLIW CPU n cycles to execute the corresponding VLIW code in parallel if no address conflict is detected. Thus, the speedup of the VLIW code is 6 approximately.

4 Theoretical Speedups

Because of the indeterminacy of parallel execution of programs, it is very difficult to precisely analyze the complexity and code space of the final VLIW code. The results we obtained are related to probabilities of events that address conflicts occur.

For the sake of clarity, assume that (*a*) all operations complete within one cycle, (*b*) all PEs share only one memory bank, and (*c*) each of PEs have a memory read unit, a memory load unit and four BRLC units. Proofs of theorems can be found in [8].

Definition 1. Let op_1 and op_2 be two operations of a program. The number of operations between op_1 and op_2 plus 1 is referred to as *operation distance*, denoted by dis(op_1, op_2).

Definition 2. Let op_1 and op_2 be two operations of a VLIW program, and operation op_1 executes before operation op_2 in the original sequential code. If op_1 and op_2 have been arranged and the number of VLIW instructions between these two operations is N, *arrangement distance* of these two operations, denoted by $d(op_1, op_2)$, is

$$d(op_1, op_2) = \begin{cases} N+1, & \text{if } op_1 \text{ executes before } op_2, \\ -N-1, & \text{if } op_1 \text{ executes after } op_2, \\ 0, & \text{otherwise.} \end{cases} \quad (1)$$

Definition 3. Let op_1 and op_2, respectively, be two ambiguous store and load operations. Let the arrangement distance $d(op_1, op_2) < 0$. When an address conflict is detected during run-time, some NOP operations are inserted to implement the postponement of the incorrect memory load operation and its successive operations. The number of inserted NOP operations is called *compensation code measure*, denoted by Ω.

Definition 4. The duration when compensation NOP operations are executed before op_2 is referred to as *pre-compensation period*, denoted by D_1. Similarly, the duration when compensation NOP operations are executed after op_1 is referred to as *post-compensation period*, denoted by D_2.

Given a loop program, an operation has different arrangement place in different iterations. The following definition presents specific arrangement information of operations in different iterations.

Definition 5. For any op_1 and op_2 belonging to a loop whose loop counter is n, suppose that $op_1^{(k)}$ and $op_2^{(j)}$ denote the k-th iteration of op_1 and the j-th iteration of op_2, respectively, where $1 \le j \le n$ and $1 \le k \le n$. If $j \ne k$, $d(op_1^{(k)}, op_2^{(j)})$ is referred to as *inter-body arrangement distance*. Otherwise, $d(op_1^{(k)}, op_2^{(j)})$ is referred to as *inner-body arrangement distance*.

Any modulo scheduling algorithm of a loop has to determine the initial interval, II, of the loop before scheduling it. That is, the modulo scheduling algorithm has to determine the inter-body arrangement distance of the first operation in two adjoining iterations, $d(op_1^{(k)}, op_1^{(k+1)})$. It is easily found that the inner-body arrangement distance of op_1 and op_2 in different iterations are the same, abbreviated as $d_{\text{inn}}(op_1, op_2)$. If op_1 executes before op_2 in the original code, $op_1^{(j)}$ executes before $op_2^{(j)}$ in the VLIW code when software pipelining algorithm is applied, *i.e.*, $d_{\text{inn}}(op_1, op_2) > 0$.

Theorem 1. Let II = 1. Suppose that l be the length of the sequential code of the loop and n be the loop counter. $op_1^{(k)}$ and $op_2^{(j)}$ are two arranged ambiguous load and store operations, respectively, and their inner-body arrangement distance is $d_{\text{inn}}(op_1, op_2) = d$. After some address conflicts have occurred, that is, the address conflict whose body difference is i has occurred j_i times, where $1 \le i \le d$ for any i, the parallel speedup of the VLIW program, called *general speedup*, is

$$S = \frac{ln}{n + 2l - 4 + \sum_{i=1}^{d} j_i \left(d - i + 1\right)}. \quad \textbf{(2)}$$

After address conflicts have occurred m times, the average value of general speedups is of the form

$$\overline{S(m)} = \frac{2ln}{2n + md + m + 4l - 8}. \quad \textbf{(3)}$$

Theorem 2. Suppose that probabilities of events that address conflicts between any two different iterations occur are independent of each other and probabilities of events that address conflicts with different body differences in an iteration occur are mutual. Let p_i be the probability of the event that an address conflict whose body difference is i, occur in an iteration, where $1 \le i \le d$ for any i. Other assumptions of the theorem are the same as those of Theorem 1. If address conflicts occur m times with probability, the compensation code measure, $\Omega_P(m)$, is related to m's probability, that is,

$$\Omega_P(m) = \sum_{\substack{j_1 + j_2 + \ldots + j_d = m \\ 0 \le j_1, j_2, \ldots, j_d \le m}} \left(\binom{n}{j_1, j_2, \ldots, j_d, n - m} \left(1 - \sum_{i=1}^{d} p_i\right)^{n-m} \prod_{i=1}^{d} p_i^{j_i} \sum_{i=1}^{d} j_i \left(d - i + 1\right) \right). \quad \textbf{(4)}$$

The corresponding parallel speedup with probability, called *probabilistic speedup*, is of the form

$$S_P(m) = \frac{l \times n}{\Omega_P(m) + n + 2l - 4}. \quad \textbf{(5)}$$

Parallel speedups of the VLIW program are different from each other when distinct address conflicts occur. Being the means of estimating speedup before program execution, the probabilistic speedup $S_P(m)$ denotes the expected value of the parallel speedup. The probabilistic speedup is an important parameter to show the efficiency of the RTPAD method.

Theorem 3. Assumptions of the theorem are the same as Theorem 2. The average value of parallel speedups when some address conflicts occur with probability, called *mean speedup with probability*, is

$$\overline{S} = \frac{ln}{\overline{\Omega} + n + 2l - 4}, \quad \textbf{(6)}$$

where $\overline{\Omega}$ is the average value of compensation code measures with probability and $\overline{\Omega} = \frac{n\sum_{i=1}^{d} p_i(d-i+1)}{1-\left(1-\sum_{i=1}^{d} p_i\right)^n}$. Convergence of mean speedup with probability is

$$\lim_{n\to\infty} \overline{S} = \frac{l}{1+\sum_{i=1}^{d} p_i(d-i+1)} . \quad (7)$$

The mean speedup with probability $\overline{S}$ denotes the average value of the parallel speedups, which is an important parameter to show the average performance of the RTPAD method. When the probability of events that address conflicts occur is 0, convergence of mean speedup with probability is l.

5 Experiment Results

This section briefly introduces and analyzes experimental results of the RTPAD algorithm for the sample code. For load operations which can possibly result in run-time address conflicts, the method inserts some RTPAD instructions before them. For the sake of the clarity, we only discuss the loop program shown in Fig. 3. More detailed experimental results and practical applications can be seen in [8].

The RTPAD method inserts three RTPAD operations before op_5. That is, there exist three probabilistic parameters, p_1, p_2 and p_3, when software pipelining is applied. Suppose that $p_1 = p_2 = p_3$, and $p_1 + p_2 + p_3 = p$. We executes the compiled code of the loop program 10,000 times repeatedly where the loop counter n = 10,000.

Fig. 4 illustrates the speedups of the compiled code shown in Fig. 3, while the p-axis denotes the probability of occurring address conflicts between two iterations, and the S-axis means the speedup. In Fig. 4 max, min, and mean denote the maximum speedup, minimum speedup, and mean speedup obtained through 10,000 times executions respectively, while $\lim S$ denotes convergence of mean speedup with probability obtained by Theorem 3. The experiment shows that the RTPAD method works very well.

6 Conclusion

This paper has proposed a method of run-time pointer aliasing disambiguation, RTPAD. Applying the RTPAD approach to a typical loop example, this paper has indicated that it has solved pointer aliasing problem with the same speed as software pipelining only applying compensation approach.

The RTPAD approach presented in this paper has its own advantages. First, it is good for irreversible code because the run-time checking method has no redo

problem. Second, the code space for compensation code is limited because any RTPAD operation only needs one RTPAD control instruction. and last, it has no rerollability problem that other run-time checking methods have.

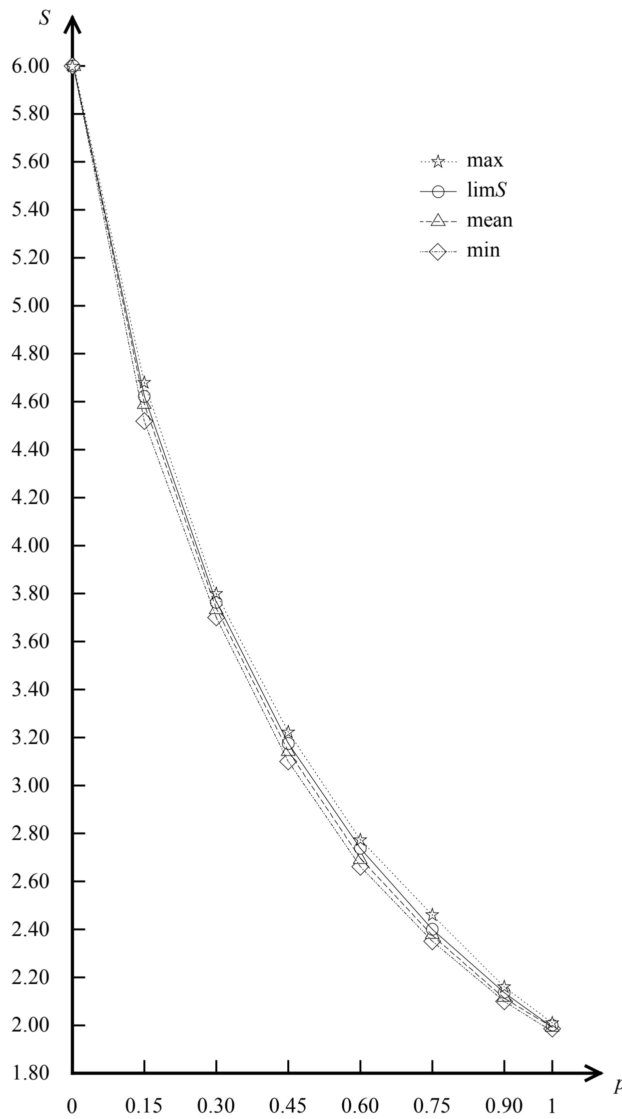

Fig. 4. Speedups of the compiled code shown in Fig. 3

In addition, this paper has theoretically described three parallel speedups of the RTPAD approach, *i.e.*, general speedup, probabilistic speedup and mean speedup with probability. Because of the indeterminacy of parallel execution of programs, it is very

difficult to precisely analyze the complexity and code space of the final VLIW code. The obtained results are related to probabilities of events that address conflicts occur. These theoretical speedups will be helpful for studying and evaluating the instruction-level parallel techniques.

Acknowledgement

This work was supported by National Nature Science Foundation, grant number 60173010, of *P. R. China*.

References

1. Rau, B. R., Fisher, A.: Instruction-Level Parallel Processing: History, Overview, and Perspective. Journal of Supercomputing 7 (1993) 9–50
2. Rong, H. B., Tang, Z. Z., Govindarajan, R., Douillet, A., Gao, G. R.: Single-Dimension Software Pipelining for Multi-Dimensional Loops. In: Proceedings of the 2nd IEEE/ACM International Symposium on Code Generation and Optimization, 21-24 Mar. 2004, San Jose, CA. IEEE Computer Society, Los Alamitos, CA (2004) 163–174
3. Qiao, L., Huang, W. T., Tang, Z. Z.: A Static Data Dependence Analysis Approach for Software Pipelining. In: Jin, H., Reed, D., Jiang, W. (eds.): Proceedings of IFIP International Conference on Network and Parallel Computing, Beijing, Lecture Notes in Computer Science. Springer-Verlag, Berlin Heidelberg New York (2005) accepted by NPC'05
4. Qiao, L., Huang, W. T., Tang, Z. Z.: Coping with Data Dependencies of Multi-Dimensional Array References. In: Jin, H., Reed, D., Jiang, W. (eds.): Proceedings of IFIP International Conference on Network and Parallel Computing, Beijing, Lecture Notes in Computer Science. Springer-Verlag, Berlin Heidelberg New York (2005) accepted by NPC'05
5. Nicolau, A.: Run-Time Disambiguation: Coping with Statically Unpredictable Dependencies. IEEE Transactions on Computers 38 (1989) 663–678
6. Su, B., Hu, E. W., Najarian, J.: Technical Description of SPLIT – A Hardware/Software Combined Approach for Run-Time Pointer Aliasing Disambiguation. Tech. Rep. 108, Department of Computer Science, William Paterson University, NJ (1996)
7. Qiao, L., Tang, Z. Z., Wang, S. Y.: Control Strategies of Software Pipelining: Dealing with the Prologue and the Epilogue of Nested Loops. In: Zhou, X., Xu, M., Lou, S., Yang, X. (eds.): Proceedings of the 3rd Workshop on Advanced Parallel Processing Technologies, 19-21 Oct. 1999, Changsha, China. Publishing House of Electronics Industry, Beijing (1999) 177–181
8. Qiao, L.: On Data Dependencies in Software Pipelining. Doctorial Dissertation, Department of Computer Science, Tsinghua University, Beijing (2001)
9. Qiao, L., Zou, H. X., Wen, Q., Tang, Z. Z.: Exploiting Instruction-Level Parallelism for the FM^{M}let Transformation. In: Ip, H. S., Shi, Y. C., Zhang, X. J., (eds.): Proceedings of the 10th Joint International Computer Conference, 4-6 Nov. 2004, Kunming, China. International Academic Publishers, Word Publishing Corporation, Beijing (2004) 587–592