HAL
open science

Load-Store Optimization For Software Pipelining

Min Dai, Christine Eisenbeis, Sid Touati

To cite this version:

Min Dai, Christine Eisenbeis, Sid Touati. Load-Store Optimization For Software Pipelining. Computer Architecture News, 2000, 28 (1), pp.3 - 10. 10.1145/346023.346027 . hal-00647085

HAL Id: hal-00647085

https://inria.hal.science/hal-00647085v1

Submitted on 1 Dec 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Load-Store Optimization For Software Pipelining

Min DAI, Christine EISENBEIS, Sid-Ahmed-Ali TOUATI
Institut National de Recherche en Informatique et en Automatique
INRIA Rocquencourt B.P.105 - 78153 Le Chesnay
email : (First.LastName)@inria.fr

Abstract

Software pipelining can generate efficient schedules for loop by overlapping the execution of operations from different iterations in order to exploit maximum Instruction Level Parallelism (ILP). Code optimization can decrease total number of calculations and memory related operations. As a result, instruction schedules can use freed resources to construct shorter schedules. Particularly, when the data is not presented in cache, the performance will be significantly degraded by memory references. Therefore, elimination of redundant load-store operations is most important for improving overall performance. This paper introduces a method for integrating software pipelining and load-store elimination techniques. Moreover, we demonstrate that integrated algorithm is more effective than other methods.

Keywords : software pipelining, load-store optimization, code generation, Instruction Level Parallelism.

1 Introduction

Software pipelining can generate efficient schedules for loops by overlapping the execution of operations from different iterations. With sufficient overlap, the software pipelined schedules can take maximum resources of machine. Modulo scheduling is a software pipelining technique which generates an arrangement of operations in a loop, so that there is no resource conflicts or data dependence violations. Traditionally, *modulo scheduling* restricts the space of software pipelined schedules by initiating consecutive loop iterations at constant rate, i.e. *Initiation Interval* (II). The initiation interval is bounded by the *minimum initiation interval* (MII) [1, 2, 3], which is a lower bound on the smallest feasible value of II. In general, MII is constrained by either resource constraints (MII_{res}) or cyclic dependence constraints (MII_{dep}).

Code optimization, such as load-store elimination, is most important in software pipelining approach. Many researches of loads/stores elimination have been proposed. *Callahan et al* [4] first introduced the concept of *register pipelining*, in which array references that access same array elements are allocated in a register pipeline in order to minimize the memory traffic. *Duesterwald* [5, 6] and *Bodik* [7] extend *Callahan's* method by using data flow analysis for detecting and eliminating redundant load-store operations. *Duesterwald's* method is based on reference variable analysis. Her method treats each array reference independently and computes the length of array live range to determine the number of shift operations. *Bodik* proposes *Congruent Class of References Analysis* which analyzes the value of array references within a *congruent class*. This method uses minimal number of memory related operations and minimal number of register-to-register copy operations in loops. But his method is only suitable for small dependance distances. Furthermore, only considering to reduce the memory traffic may not

significantly improve the overall performance, since we reduce the number of load operations instead of increasing the number of shift operations. The generated schedule can not exploit more ILP while the transformed code needs more resources. In order to exploit more ILP, we developed a method [8] which minimizes load-store operations after performing software pipelining. This method should also introduce shift operations to replace load operations. In this paper, we integrate software pipelining and load-store optimization techniques in order to avoid shift operations and exploit maximal ILP in loops.

2 Background

2.1 Constraints affecting loop scheduling

To generate a software pipelined schedule, we require first an estimate of the initiation interval (II) which is the length of new pipelined loop. Choosing the minimum feasible II achieves the highest possible steady-state performance. To determine the *minimum initiation interval* (MII), both dependence constraints and resource constraints should be taken into account respectively [9].

The resource usage imposes a lower bound on the initiation interval (MII_{res}), which is computed by accounting for all the resources consumed by each operation of an iteration. For example, if a resource is used u times on a machine with v identical copies of u, therefore, MII cannot be smaller than u/v, because of the modulo scheduling constraint.

Another factor that is used for an estimate of the lower bound on the initiation interval is cyclic dependence. To characterize the dependences, a dependence edge can be annotated with (λ, δ) pair. The λ indicates the number of iterations the dependence spans, and δ indicates the time that must elapse between the time the first operation is issued and the time the second operation is issued. For cyclic dependence, a cycle θ contains a series of edges, we use δ_θ to present the sum of the δ and λ_θ to present the sum of the λ on θ. The dependence constrained lower bound, MII_{dep}, is determined by accounting for all the elementary cycles in data dependence graph that are created by recurrences. In general, a cyclic path must satisfy the following dependence constraint inequality.

$$\sigma(a,i) + \delta_\theta(e) \leq \sigma(a, i + \lambda_\theta(e)) \;\; \forall \; cycle \; \theta \quad (1)$$

Since $\sigma(a,i)$ denotes the execution cycle where the instance of operation a of ith iteration is issued. $\sigma(a, i+\lambda_\theta(e))$ can be represented with $\sigma(a,i)+II*\lambda_\theta(e)$. The above formula can be rewritten as:

$$\delta_\theta(e) - II * \lambda_\theta(e) \leq 0 \;\; \forall \; cycle \; \theta \quad (2)$$

The minimum initiation interval can be found by solving for the minimum value of II in above inequality. Therefore, MII_{dep} is computed by taking the maximum value of $\frac{\delta_\theta}{\lambda_\theta}$ for all cycles.

$$MII_{dep} = max_{(\forall \; cycles \; \theta)} \left\lceil \frac{\delta_\theta}{\lambda_\theta} \right\rceil \quad (3)$$

Finally, the actual lower bound on the initiation interval is then:

$$MII = max(MII_{dep}, MII_{res}) \quad (4)$$

Any cyclic path having $\frac{\delta_\theta}{\lambda_\theta}$ equal to MII is termed a critical cycle.

2.2 Longest path - critical cycle

In a loop schedule, each dependence edge (a,b) is assigned a weight of $\delta - II * \lambda$ in order to satisfy the dependence constraints. For a pair of nodes on path P, the minimum schedule distance can be computed by: $D_P = \delta_P - II * \lambda_P$. If P is a cyclic path, we have a definition:

Definition 2.1 *At a node v in DDG, for all cyclic paths θ passing through v, we have the*

schedule distance $D_\theta(v,v) \leq 0$. If a path θ_s passes v and the schedule distance $D_{\theta_s}(v,v) \geq D_\theta(v,v)$, the path θ_s is called the longest path.

In loop scheduling, we would like to compute the minimum distance between two nodes. By the definition 2.1, the schedule distance of cyclic path is always zero or negative. Therefore, by reversing the sense of inequalities, the longest path is equivalent to the smallest schedule distance. For the example of figure 2(c), we can compute MII_{dep} for cyclic dependence. Supposing each operation can be issued in one unit cycle, table 1 gives the results of the computation for some cyclic paths. By Equation 3, we have: $MII_{dep} = max(\lceil\frac{4}{1}\rceil, \lceil\frac{3}{2}\rceil, \lceil\frac{4}{3}\rceil) = 4$.

Cyclic Path (θ)	(λ, δ)	$\lceil\frac{\delta}{\lambda}\rceil$	D_θ
$1 \to 3 \to 6 \to 7 \to 1$	(1,4)	$\lceil\frac{4}{1}\rceil$=4	0
$2 \to 3 \to 4 \to 2$	(2,3)	$\lceil\frac{3}{2}\rceil$=2	-5
$2 \to 3 \to 6 \to 7 \to 2$	(3,4)	$\lceil\frac{4}{3}\rceil$=2	-8
$5 \to 6 \to 7 \to 5$	(2,3)	$\lceil\frac{3}{2}\rceil$=2	-5

Table 1: *Computing II for cyclic dependence graph*

The distances corresponding to each cyclic path are presented in the D_θ column. We can find that the path $(1,3,6,7,1)$ is the longest path, which dominates MII_{dep}. The others can be ignored. Therefore, computing cyclic dependence constraints is equivalent to finding the longest path in the dependence graph.

Definition 2.2 *Let G = (N, E, λ, δ) be a loop dependence graph, if there exists a set of cyclic paths Θ, the minimal initiation interval due to dependence constraints is determined by the longest path θ_s (or critical path). The other cycles are redundant for computing MII_{dep}.*

3 Load-store elimination

3.1 Eliminating redundant loads

To determine whether a load operation can be eliminated we can use the informations of data dependence. In data dependence graph, if there exists a predecessor of a load operation, the loaded value is available in the predecessor. Therefore, the undesirable load can be precisely characterized as follows:

Definition 3.1 *A load operation is redundant if there exists a predecessor of this load, i.e. the load is true or input dependent on the predecessor.*

There is two types for eliminating redundant loads: true dependence elimination and input dependence elimination. The rules of redundant load elimination are presented in Rule 3.1 and Rule 3.2.

Rule 3.1 (True dependence elimination) *A load can be safely eliminated from true dependence graph, if there exists a predecessor (a store). While deleting the load, a true dependence edge should be connected from the predecessor of store to the successor of load. This edge is weighted by (λ', δ'), where δ' is equal to the latency of the predecessor of store, λ' is computed as follows:*

$$\begin{cases} \lambda' = \sum_{e \in P} \lambda(e) \\ P: \textit{path from pred. of store to succ. of load} \end{cases}$$

Rule 3.2 (Input dependence elimination) *A load can be eliminated, if there exists an input dependence edge. Deleting one of loads, a true dependence edge must be added from remainder load to the successor of eliminated load. The edge is weighted by (λ', δ'), where δ' is equal to the latency of the load, λ' is computed as follows:*

$$\begin{cases} \lambda' = \lambda(e1) + \lambda(e2) \\ \textit{e1: an input dep. edge between two loads} \\ \textit{e2: a true dep. edge between the deleted} \\ \qquad \textit{load to its successeor} \end{cases}$$

3.2 Eliminating redundant stores

A redundant store operation can be characterized as follows:

Definition 3.2 *A store is dead if a later store causes the earlier store to become redundant along a program path, that is, there exists a successor of the store in output dependence graph.*

The method of output dependence elimination is characterized by the Rule 3.3.

Rule 3.3 (Output dependence elimination) *A store can be eliminated if an output dependence edge is detected. While deleting dead stores for array references, we should insert the last instances of store in epilogue. If the last instances are never used, we can also ignore these store instances.*

3.3 Reducing MII

Now, we present that elimination of load can optimize software pipelined schedules and reduce the length of MII. We know that MII_{dep} is determined by the cyclic dependence. In section 2, we have proven that only the longest path is considered to compute MII_{dep}. Thus, supposed that Figure 1(a) is a longest path θ_s for a loop. MII_{dep} is computed by using the equation 3.

$$MII_{dep} = \frac{\delta_{\theta_s}}{\lambda_{\theta_s}} = \frac{\delta_1 + \sum \delta_i + \delta_3 + \delta_4}{\lambda_1 + \sum \lambda_i + \lambda_3 + \lambda_4}$$

After eliminating redundant load by rule 3.1, the dependence graph is shown in Figure 1(b). The minimum initiation interval (MII'_{dep}) corresponding to the cyclic path θ'_s is computed as:

$$MII'_{dep} = \frac{\delta'_{\theta_s}}{\lambda'_{\theta_s}} = \frac{\sum \delta_i + \delta_3}{\sum \lambda_i + \lambda_1 + \lambda_3 + \lambda_4}$$

As the value of δ is always positive, we can find that after eliminating redundant load, the new minimum initiation interval (MII'_{dep}) is always smaller than the original (MII_{dep}) .

Furthermore, by eliminating a redundant operation, instruction schedules can use freed hardware resources to construct shorter schedule. Therefore, if both load and store are on the critical path, the length of pipelined schedules can be essentially reduced.

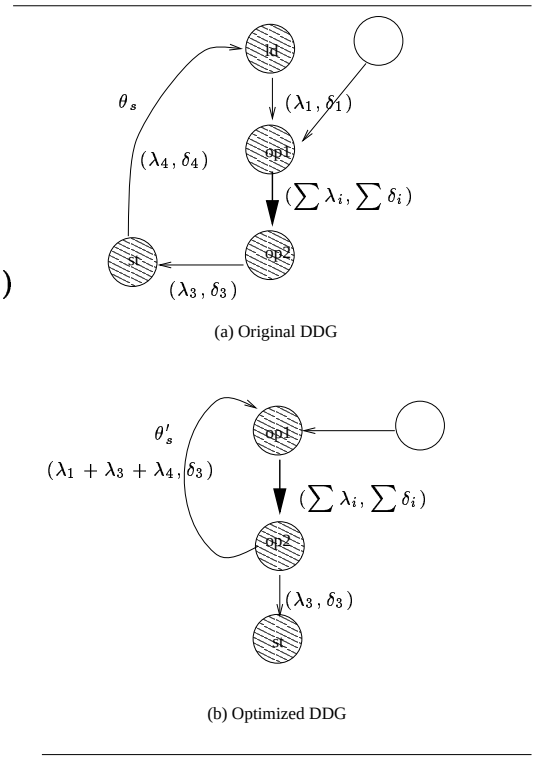

Figure 1: Elimination of loads for reducing MII

4 Algorithms

This section presents the application of integrating software pipelining and memory operation optimization techniques. First, the optimization algorithms are an attempt to minimize the load-store operations in loops, which consist 2 steps: eliminating loads for true and input dependence; and eliminating stores for output dependence. Then, code generation steps are used for realizing loop transformations.

Before describing the optimization algorithms, we should first introduce some definitions.

Definition 4.1 *If $u \rightarrow v$ is an edge in a dependence graph, u is a predecessor of v, and v is a successor of u. We use $Succ(u)$ and $Pred(v)$ to denote the set of all successors of u and the set of all predecessors of v, $d(u,v)$ and $c(u,v)$ to denote the distance vector and the condition*

of dependence edge $u \rightarrow v$.

Definition 4.2 *Let S be the set of nodes in a loop, and R be the set of scalar and array references, we have the variables:*

- Def(n)$\in$ R is a definition reference of node n
- Use(n) $\in$ R is an use reference of node n
- LOAD $\subset$ S is a subset of nodes corresponding to load operations
- STORE $\subset$ S is a subset of nodes corresponding to store operations

4.1 Eliminating loads/stores before software pipelining (ELSBSP)

4.1.1 Eliminating loads

In order to avoid a redundant load, we implement the rules of *true dependence elimination* and *input dependence elimination*. We refer to a point at which an operation is initialized as a *load point*, our approach treats only those *load points*. If the *load point* has at least a predecessor in true and input dependence graph, we call this point as *elimination point*. Then, no load is needed at this point, because the loaded value has already been available in a register.

4.1.2 Eliminating stores

The process of store elimination so that a store can be avoided is to find a *store point* as *killing point* at which the store operation is killed by a later store. While elimination of loads requires to find backward points at *elimination points*, the elimination of stores requires to find forward points at *killing points*. The above algorithms is summarized in algorithm 3.1.

Algorithm 3.1. Load-store elimination :

Eliminating loads for true dependence :

```
compute true dependence for loop
find the set of load, LOAD, in innermost loop (I)
for (∀ n ∈ LOAD)
  if (∃ Pred(n)) then
    for (∀ m ∈ Pred(n))
      find the smallest distance d_min(m,n)
    for (∀ l ∈ Succ(n))
      Use(l) ← Use(m)
      change index I of Use(l) by I − d_min(m,n)
    endfor
    delete load n by the rule 3.1
  endif
endfor
construct new data dependence graph
```

Eliminating loads for input dependence :

```
compute input dependence for loop
find the set of load, LOAD, in innermost loop (I)
for (∀ n ∈ LOAD)
  if (∃ Pred(n)) then
    for ( ∀ m ∈ Pred(n))
      find the biggest distance d_max(m,n)
    for (∀ l ∈ Succ(n))
      Use(l) ← Def(m)
      change index I of Use(l) by I − d_max(m,n)
    endfor
    delete load n by rule 3.2
  endif
endfor
construct new data dependence graph
```

Elimination of stores for output dependence :

```
compute output dependence for loop
find the set of store, STORE
for (∀ n ∈ STORE)
  if (∃ Succ(n) ) then
    for (∀ m ∈ Succ(n))
     find the biggest distance d_max(n,m)
    endfor
    weight the node m by p(d_max(n,m), Pred(n))
    delete all other successors l ∈ Succ(n) l ≠ m
    delete store n by rule 3.3
  endif
endfor
construct new data dependence graph
```

4.2 Code Generation

Generally, after eliminating redundant load-store operations, many methods can be directly implemented for generating pipelining schedule [3, 2, 8].

Unfortunately, these existing methods do not consider the initial value for each array reference. Therefore, before using software pipelining, we should decide which value an array reference read comes from. For example of figure 2(d), there exists a dependency from node 6 to node 3 with the distance 1 and a condition: c(a,b) = if (I > 3), that is if (I > 3) the read value by T5[I-1] is computed in the preceding iteration, otherwise the read value is not defined in the loop. It should be initialized in the prologue. So in our approach, we should first treat all conditions of each dependence edge and then parallelize the kernel loop. Our code generation algorithm is presented by the following steps.

4.2.1 Initializing array reference in prologue and epilogue

The first step consists of determining the initialization values for array references. It is well known that *software pipelining* is based on *modulo scheduling* method. Modulo scheduling selects a schedule for one iteration of the loop such that, when that schedule is repeated, no resource and dependence constraints are violated. Once a pipelining schedule has been found for loop body, we need to add a prologue and epilogue for completing entire loop, (more details in [8]).

Initializing prologue code requires the precise data dependence informations to determine initialization values for all array references. We have shown that data flow analysis can give us these precise informations, as well as the last write reference for a read operation. Algorithm 3.2 determines the initial values of array references in prologue and inserts last store instances in epilogue by using the precise data dependence informations.

4.2.2 The number of unrolling / Software Pipelining loop

In order to obtain full parallelism, a kernel loop would be unrolled the maximum number of *live range*, however this will increase the register pressure. Considering the number of registers in the target machine, we should choose a *suitable* number of unrolling. If the number of unrolling is smaller than *live range*, we should insert *shift* operations to keep the values in registers for reuse.

In our approach, we use DESP algorithm to find a pipelining schedule. The position of each operation is fixed in a matrix which present an operation by row number (rn) and column number (cn). The live range of each variable is determined by checking the row-number and column-number. For example, if a variable is created by operation op_1 scheduled at $cn(op_1)$ and $rn(op_1)$, and the last use of this variable is located at $cn(op_2)$ and $rn(op_2)$. Thus the live range (LR) is equal to:

$$LR = cn(op_1) - cn(op_2) + u$$
$$u = \begin{cases} 1 & rn(op_1) < rn(op2) \\ 0 & otherwise \end{cases}$$

The unrolling number is determined by maximum live range of variables.

Algorithm 4.1. Initializing prologue and epilogue:

initializing prologue:

Let S_{P_r} be the set of instances in prologue
Let op_i^k be an instance of op_i at k iteration
for ($\forall$ op_i^k)
  if ($\exists$ (op_i δ op_j and $op_j \in S$)) **then**
    compare $c(op_i, op_j)$ with k
    if (false) **then**
      initialize reference $Def(op_j)$ in prologue
  endif
endfor
for ($\forall$ $op_i^k \in S_{P_r}$)
  change index of array references in op_i by k
  add this transformed operation in prologue
endfor

initializing epilogue:

Let S_{P_o} be the set of instances in epilogue
for ($\forall$ op_i $\in S$)
  if (op_i $\in STORE$) **Then**
    check weighted value of operation op_i, $p(v, s)$
    if ($v > 0$) **then**
      find operation s associated with a store
      For (i = 1 to v)
        add s in the end of epilogue
        insert a store for s
      endfor
    endif
  endif
endfor
for ($\forall$ op_i^k $\in S_{P_o}$)
  change index of array references in op_i by k
  add this transformed operation in epilogue
endfor

4.2.3 Temporary reference replacement - Register allocation

The next step in code generation is to create scalar variables to replace temporary array reference to perform data reuse. An approach of register allocation for subscripted variables was developed by Callahan, *et al.* [4]. This approach which is used in our technique, begins by calculating the number of scalar variable τ_i, needed to hold each variable generated by temporary array reference $T_i[I]$, which associates with the maximum dependence distance (d), i.e., τ_i = max(d) + 1. For example in Figure 2(e), the temporary array reference $T_3[I]$ needs three variables which call R_{10}, R_{11} and R_{12}.

The finally result for our example is shown in figure 2(f). Here, the ";" delimiter separates operations that are executed in parallel.

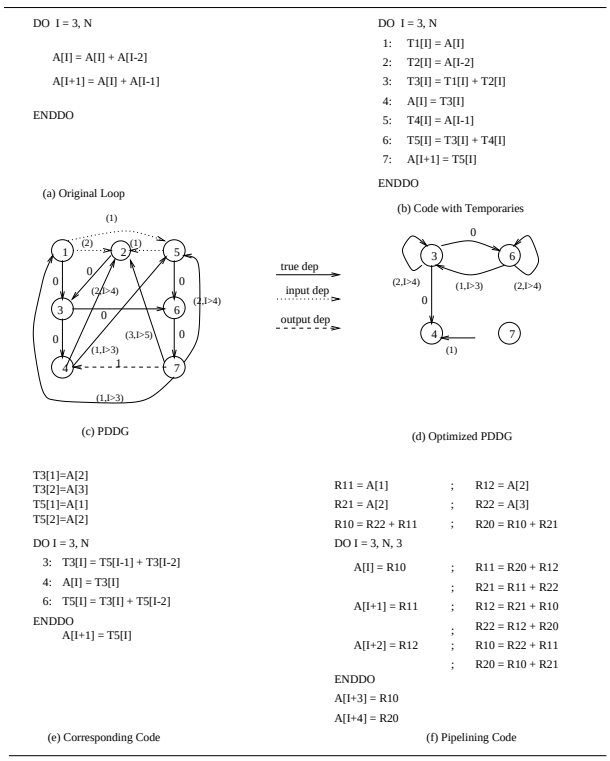

Figure 2: Loop Example

5 Experimental Results

In this section, we report our experimental results by using our optimization method. The source Fortran files are from Livermore benchmark suite. We give experimental results that demonstrate the effectiveness of load-store redundant elimination.

Loop	Org.	Bodik		ELSBSP	
	TNL	TNL	P	TNL	P
L1	1200	800	33%	800	33%
L5	3000	2000	33%	2000	33%
L7	1080	360	66%	360	66%
L11	2000	1000	33%	1000	33%
L12	2000	1000	33%	1000	33%
L23	50000	40000	20%	40000	20%

Table 2: Load number before and after optimization

In table 2, we give the dynamic count of loads before and after optimization, where **TNL** is total number of loads, and **P** is percentage of elimination. For comparing our method ELSBSP (Elimination of Load and Store Before Software Pipeline) with [7], we defined the same iteration numbers. The number of loads due to array references was determined by examining the assembly code output of the compiler. We test only some number of Livermore loops. Many loops did not contain opportunities for pipelining or contain inner-loop conditional control flow which we do not handle. As we can see, our methods can eliminate the same number of loads as Bodik's method. But it is more effective than other methods for improving the performance. Existing methods eliminate all redundant load operations, but introduce shift operations to keep the value for reuse. On the contrary, our method first eliminates redundant loads, then we use software pipelining to generate a pipelining schedule and unroll the pipelined kernel loop. Therefore no shift operations are needed.

Figure 3(a) shows the performances of tested loops. Each loop is compiled and executed at -O2 optimization level of Fortran compiler on UltraSPARC processor. The performance is represented by the execution time. Figure 3(b) presents the execution time of loop23 on DEC-Alpha processor for different loop scheduling and code optimization approaches. From left to right, the continuous bars represent the execution time of source program, Bodik's method, software pipeline heuristic, ELSASP [1] heuristic, and ELSBSP heuristic. We

[1] Elimination of Load and Store After Software

can see that ELSBSP heuristic reduces the execution time significantly. Comparing the execution time of other methods, on -O2 or -O3 optimization level of Fortran complier, implementation of ELSBSP can obtain best improvement of performance. In this example, ELSBSP heuristic can improve the performance till 30%. These results present that elimination of load-store is more successful while integrating with software pipelining.

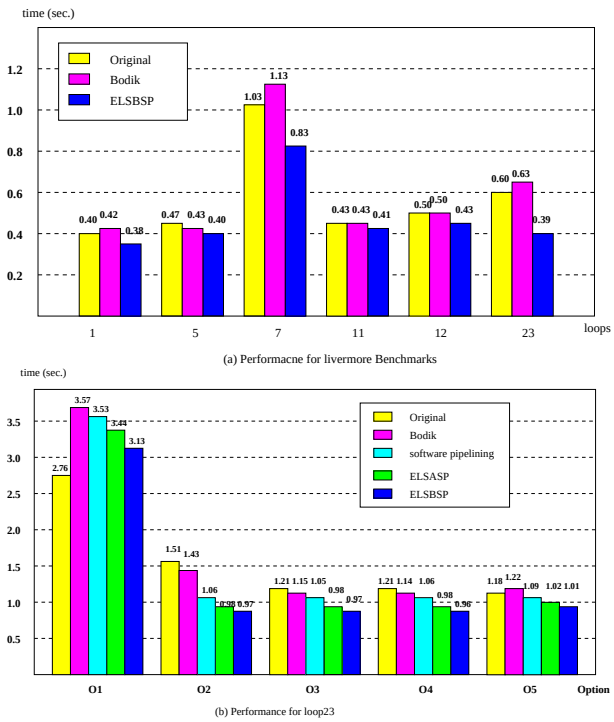

Figure 3: Execution time of Livermore Benchmark loops on Ultra-SPARC processor

6 Conclusion

In this paper, we introduce a new method for eliminating redundant load and store by using precise data dependence informations. Eliminating redundant loads can improve both MII_{dep} and MII_{res} and result in reducing the length of pipelined schedules. The effect of these optimizations is to expose new opportunities for minimizing memory accesses. Therefore, by eliminating redundant load-store operations, we can improve overall performance of an application. The experimental results also present that integrating software pipelining and load-store elimination is more efficient than other optimization techniques.

Pipelining [8]

References

[1] B. R. Rau and J. A. Fisher. Instruction-level parallel processing : History, overview, and perpective. *The Jornal of Supercomputing*, 7:9–50, 1993.

[2] V.H. Allan, R.B. Jones, R.M. Lee, and S.J. Allan. Software Pipelining. *ACM Computing Surveys*, 27(3):367–432, September 1995.

[3] J. Wang, C. Eisenbeis, M. Joudan, and B. Su. DEcomposed Software Pipeline : a new persective and a new approach. *International Journal on Parallel processing*, 22(3):357–379, 1994.

[4] D. Callahan, S. Carr, and K. Kennedy. Improving register allocation for subscripted variables. *Proc. of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 53–65, June 1990.

[5] E. Duesterwald, R. Gupta, and M. L. Soffa. Register pipelining : An integrated approach for scalar and subscripted variables. *International Workshop on Compiler Construction*, pages 192–206, October 1992.

[6] E. Duesterwald, R. Gupta, and M.L. Soffa. A practical data flow framework for array reference analysis and its application in optimizations. *Proc. of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 68–77, June 1993.

[7] R. Bodik and R. Gupta. Optimal placement of load-store operations for array access in loops. Technical Report DCS 95-03, University of Pittsburgh, 1995.

[8] Min Dai and Christine Eisenbeis. Source to source software pipelining. *Parallel and Distributed Computing and Networks (PDCN'97)*, August 1997.

[9] B.R. Rau and C.D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture fo high performance scientific computing. *Proc. of the 14th Conference on Microprogramming and Microarchitecture*, pages 183–198, 1981.