Tetris: A New Register Pressure Control Technique for VLIW Processors

Weifeng Xu and Russell Tessier

Department of Electrical and Computer Engineering
University of Massachusetts
Amherst, MA 01003
{wxu,tessier}@ecs.umass.edu

Abstract

The run-time performance of VLIW (very long instruction word) microprocessors depends heavily on the effectiveness of its associated optimizing compiler. Typical VLIW compiler phases include instruction scheduling, which maximizes instruction level parallelism (ILP), and register allocation, which minimizes data spills to external memory. If ILP is maximized without considering register constraints, high register pressure may result, leading to increased spill code and reduced run-time performance. In this paper, a new register pressure reduction technique for embedded VLIW processors is presented to control register pressure prior to instruction scheduling and register allocation. By modifying the relative ordering of operations, this technique restructures code to better reduce spills. Our technique has been implemented in Trimaran, an academic VLIW compiler, and evaluated using a series of VLIW benchmarks. Experimental results show that, on average, our algorithm reduces dynamic spills and improves overall cycle counts by 6% for a VLIW architecture with 8 functional units and 32 registers versus previous spill code reduction techniques.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors - compilers, optimization

General Terms Algorithms, Performance

Keywords Register Pressure, Instruction Level Parallelism, Very Long Instruction Word (VLIW) Processor.

1. Introduction

VLIW architectures exploit instruction level parallelism by performing multiple operations per clock cycle based on a fixed schedule generated by a compiler. To simplify hardware requirements, VLIW processors do not provide specialized hardware to support dynamic scheduling or out of order execution. This optimization increases the importance of accurate and efficient application mapping.

The compilation process for VLIW processors is divided into several phases including instruction scheduling and register allocation [10]. Instruction scheduling tries to maximize ILP by scheduling as many instructions as possible in parallel, which may require a large number of registers to hold variables generated in the schedule. Register allocation assigns a physical register to each variable. If a given schedule requires more registers than available physical registers, variables must be spilled to memory, regardless of the register allocation algorithm. Since each spill requires multiple time-consuming memory store/load operations, large amounts of spill code can significantly degrade performance. As a result, register pressure must be evaluated and controlled for VLIW processors.

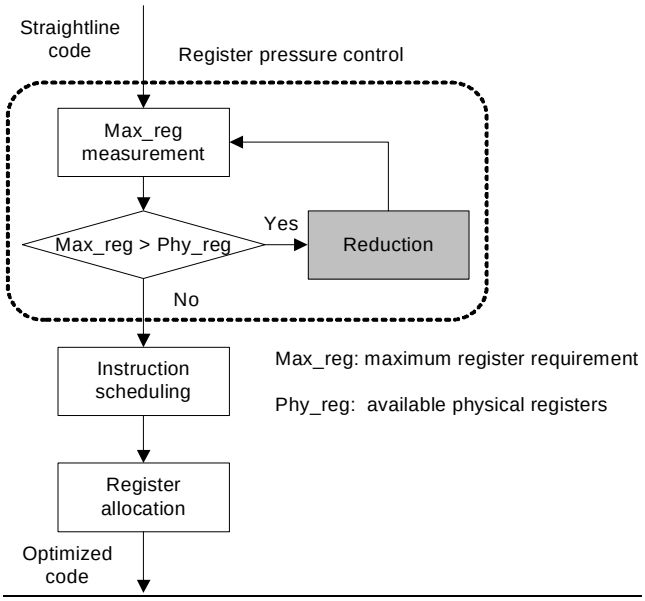

Figure 1. VLIW compilation flow with register pressure control

A number of spill reduction techniques [11, 15, 2], including register pressure control [2], have been developed. As shown in Figure 1, register pressure control is based on a measure-and-reduce methodology, which can be applied before instruction scheduling and register allocation. Using data dependencies, the measure step estimates the maximum register requirement (Max_reg) of all possible straightline code schedules. If Max_reg exceeds the number of available physical registers, Phy_reg, a reduction step is used to reduce Max_reg to Phy_reg. The reduction step allows subsequent instruction scheduling to focus on improving parallelism.

In this paper, we present a new register pressure control algorithm based on the reordering of operations. Our technique can si-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

LCTES'07 June 13–16, 2007, San Diego, California, USA.
Copyright © 2007 ACM 978-1-59593-632-5/07/0006…$5.00

multaneously serialize multiple variables to reduce Max_reg and the number of corresponding spills. To demonstrate its benefit, we have integrated the algorithm into an academic VLIW compiler, Trimaran [6]. As shown in Section 5, our technique can achieve a 38% reduction in spills and 6% improvement in performance (execution cycles of benchmarks) for a VLIW architecture with 8 functional units and 32 registers compared with previous approaches [13, 19].

The remainder of this paper is organized as follows. In Section 2, a brief discussion of previous work is presented. Section 3 provides background information and discusses the limitations of previous techniques. Our new algorithm is discussed in Section 4. Our experimental approach and results are presented in Section 5. Section 6 provides a summary of our findings and offer directions for future research.

2. Related Work

Previous work in spill code reduction can be put into several categories, general register allocation, schedule-sensitive register allocation, register-sensitive instruction scheduling and register pressure control.

General register allocation aims to minimize the number of spills based on a given schedule. Graph coloring-based register allocation [3, 4, 5] is one of the most effective register allocation approaches. Graph-based algorithms build an interference graph based on a given schedule, in which each node represents a variable and an edge between two nodes indicates that two variables cannot share the same physical register. If there are not enough physical registers to hold all variables in the interference graph, spill code must be inserted to transfer some variable storage to memory. The frequency-based live-range splitting (FBS) technique [13] attempts to isolate spill code reduction to frequently executed (hot) program regions to improve program performance. FBS uses execution frequency information to guide the splitting of variable live ranges (lifetimes) during coloring. By splitting variable live ranges in regions with lower frequency first, the overall number of spills can be reduced.

Schedule-sensitive register allocation techniques [15, 16] are applied to straightline code before instruction scheduling. In these approaches, a parallel interference graph (PIG) is created to represent all possible live range interference. To reduce the negative impact on achievable schedule length during subsequent instruction scheduling, schedule-sensitive heuristics insert a minimum number of false dependencies between variables to reduce variable interference. Govindarajan presented another early register allocation technique, the minimum register instruction sequence (MRIS) [12]. MRIS applies a special interference graph, where each node represents an instruction lineage, a series of variables that can share the same physical register.

Register-sensitive instruction scheduling controls register requirements during instruction scheduling, which is performed before register allocation. Goodman and Hsu [11] presented an integrated pre-pass scheduling (IPS) approach, where two scheduling heuristics are combined. During scheduling, register requirements are dynamically evaluated. Based on the analysis, the scheduler can switch scheduling heuristics. One heuristic improves ILP and the other heuristic reduces register usage.

Unlike the above approaches, register pressure control can be applied before both instruction scheduling and register allocation. Berson presented a technique called unified resource allocation (URSA) to reduce register pressure so that the schedule generated in subsequent instruction scheduling does not overuse registers [1, 2]. This measure-and-reduce methodology is shown in Figure 1. Experimental results [2] show that register pressure control outperforms both schedule-sensitive register allocation (PIG) and register-sensitive instruction scheduling (IPS). Touati presented several new heuristics based on Berson's measure-and-reduce approach to further improve register pressure control [18, 19]. These extensions are discussed in Section 3.

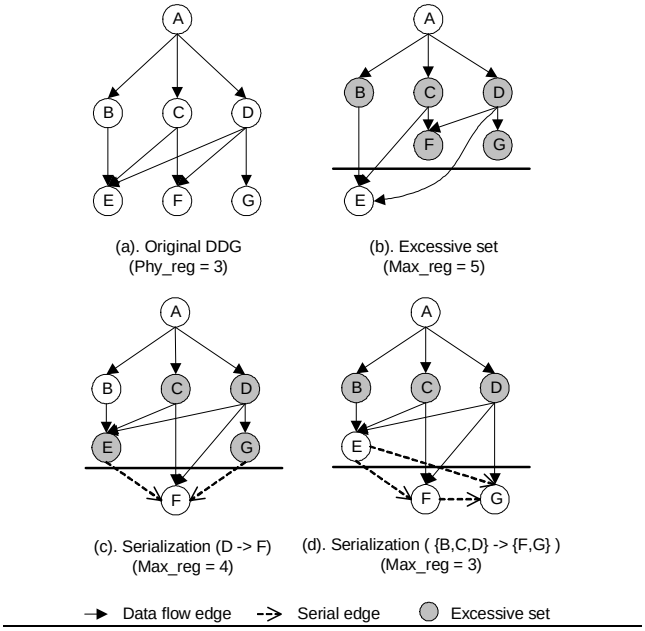

Figure 2. Register pressure reduction via variable serializations

3. Background

In this section, we first present several basic definitions. After discussing the limitations of several previous techniques [18, 19], our performance-enhancement algorithm is presented.

As shown in Figure 2-a, data dependencies of the input code can be represented in a data dependency graph (DDG), $G(V, W, E)$. In a DDG, a node v represents an operation that defines variable v and a directed edge $e(v_s, v_t)$ represents a data flow from node v_s to node v_t. The weight w of edge $e(v_s, v_t)$ represents the latency of node v_s. For demonstration purposes, all edge weights in subsequent examples are set to one. Additionally, the following notation is used.

- **Def(Var)** represents an **operation** that defines variable Var. For example, $Def(B)$ in Figure 2-a represents an operation that defines variable B.
- **Use(Var)** represents an **operation** that uses variable Var. For example, Figure 2-a has three $Use(A)$, $Def(B)$, $Def(C)$ and $Def(D)$.
- **Lv(Var)** represents the **live range** of variable Var. Lv(Var) is the distance from $Def(Var)$ to the last $Use(Var)$, where variable Var is alive. As shown in Figure 2-a, the live range of variable B, $Lv(B)$, is from $Def(B)$ to $Def(E)$, the only $Use(B)$.
- **Pred(Var)** represents a **predecessor variable**, which is required to generate variable Var. For example, Figure 2-a shows that there are three $Pred(E)$, variable B, variable C and variable D.
- **Succ(Var)** represents a **successor variable**, which is generated using variable Var. For example, Figure 2-a shows that there are three $Succ(A)$, variable B, variable C and variable D.

As discussed in Section 1, register pressure control is based on a measure-and-reduce methodology. The maximum register requirement is measured first. If it exceeds the number of available physical registers, a reduction step is applied.

For a given DDG, the maximum register requirement is the largest number of variables alive simultaneously. Since the instruction schedule is not fixed until the instruction scheduling phase, the maximum register requirement is estimated using the data dependencies of straightline code. In [1], it was shown that the maximum register requirement of a given DDG can be estimated by applying a minimum chain decomposition based on the Dilworth algorithm [8]. In this paper, we use an improved estimation technique called register saturation [19]. Previous results show that the estimated result is within one register of the measured maximum register requirement [19].

Using register saturation [19], the maximum register requirement of the DDG in Figure 2-a is 5. As shown in Figure 2-b, if $Def(E)$ is scheduled last, 5 variables $\{B, C, D, F, G\}$ are alive simultaneously since variables $\{B, C, D\}$ are required by $Def(E)$ and $\{F, G\}$ are output variables. If there are less than 5 physical registers, $\{B, C, D, F, G\}$ becomes an excessive set, which is defined below:

- An **excessive set** is a maximum set of variables in a DDG which can be alive simultaneously. The size of the set (Max_reg) exceeds the number of available physical registers (Phy_reg).

To reduce the size of the excessive set, live ranges of variables in the excessive set must be separated. In general, separating the live ranges of two variables u and v can be achieved by serialization $(u \rightarrow v)$ or serialization $(v \rightarrow u)$, which is defined below:

- **Serial edge** ($Use(u)$ to $Def(v)$) represents a directed edge from operation $Use(u)$ to operation $Def(v)$, which forces an ordering such that $Def(v)$ cannot be scheduled earlier than $Use(u)$.
- **Serialization** $(u \rightarrow v)$ represents that variable v is created after the live range of variable u ends. This can be accomplished by adding serial edges (all $Use(u)$ to $Def(v)$).

It has been proven [18] that the problem of achieving a maximum reduction using serializations is NP-hard. To address this problem, Touati [18] presented a greedy serialization technique, which evaluates all possible serializations between any pair of variables and selects the one which can best reduce the register requirement while increasing the critical path the least.

To reduce the excessive set $\{B, C, D, F, G\}$ in Figure 2-b, greedy serialization selects serialization $(D \rightarrow F)$ as shown in Figure 2-c. To force an ordering so that $Def(F)$ cannot be scheduled earlier than any $Use(D)$, two serial edges ($Def(E)$ to $Def(F)$) and ($Def(G)$ to $Def(F)$) are inserted into the original DDG. After applying serialization $(D \rightarrow F)$, Max_reg of the augmented DDG in Figure 2-c is reduced from 5 to 4. The new excessive set is $\{E, C, D, G\}$, where $\{C, D\}$ are required by $Def(F)$ and $\{E, G\}$ are output variables. Due to serial edges ($Def(E)$ to $Def(F)$) and ($Def(G)$ to $Def(F)$), the critical path changes from A-B-E to A-B-E-F.

A limitation of greedy serialization is that only a single best-cost serialization can be selected at one time, which often leads to poor performance. As shown in Figure 2-c, the excessive set $\{E, C, D, G\}$ in the augmented DDG can no longer be reduced because serialization $\{D \rightarrow F\}$ prevents other serializations.

To address this problem, a better reduction can be achieved by considering multiple variable serializations simultaneously, serializations $(set1 \rightarrow set2)$, which is defined below:

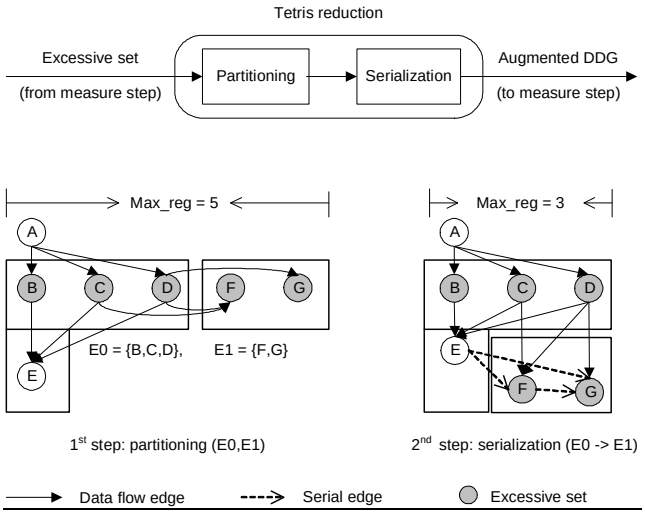

Figure 3. Tetris reduction technique

- **Compatible serializations** represent a set of serializations that can be applied together without creating cycles. A cycle caused by serial edges can make scheduling impossible.
- **Serialization** $(set1 \rightarrow set2)$ represents the maximum set of compatible serializations $(u \rightarrow v)$, where u/v is a variable in set1/set2.

As shown in Figure 2-d, serializations $(\{B, C, D\} \rightarrow \{F, G\})$ contain 5 compatible serializations: $\{B \rightarrow F\}$, $\{C \rightarrow F\}$, $\{D \rightarrow G\}$, $\{C \rightarrow G\}$ and $\{B \rightarrow G\}$. Serialization $\{D \rightarrow F\}$ is not selected because it is not compatible with serializations $\{D \rightarrow G\}$ and $\{C \rightarrow G\}$. A detailed discussion regarding compatibility checking is presented in Section 4.3. By applying these 5 compatible serializations, the maximum register requirement is reduced from 5 to 3, which is one register less than the value achieved by greedy serialization. The new maximum set in the augmented DDG is $\{B, C, D\}$, in which all variables are required by $Def(E)$.

In order to select and serialize multiple variables simultaneously for the best reduction, we present a new reduction technique, called Tetris, in the next section.

4. Tetris Reduction

4.1 Overview

The basic idea of Tetris reduction originates from the popular computer puzzle game. In a Tetris game, players try to move and place given random blocks to fit into a fixed width constraint. Similarly, Tetris reduction tries to identify blocks (subset of variables) with suitable topologies and move them to reduce the size of the excessive set from Max_reg to Phy_reg (fixed width). The similarity can be observed in Figure 3, which uses the example in Figure 2-d.

As shown in Figure 3, Tetris reduction includes two steps, partitioning and serialization.

1. **Partitioning**: this step identifies candidates for serialization. Variables in the excessive set are partitioned into two subsets, $E0$ and $E1$ based on two criteria. The first criterion indicates whether variable serializations from $E0$ to $E1$ are possible. The second criterion indicates how much register count reduction can be achieved. A detailed description of the partitioning algorithm is presented in Section 4.2.
2. **Serialization**: this step is applied to serialize variables in $E1$ after variables in $E0$ by inserting serial edges into the DDG.

The ordering of variable serializations is decided such that a maximal set of variable serializations can be applied. The detailed serialization algorithm is discussed in Section 4.3.

4.2 Partitioning

4.2.1 Definitions

Before discussing partitioning in detail, additional definitions are presented:

- **NSE(u,v)** is a directed non-serializable edge (NSE) from variable u to v. The edge indicates that variable v cannot be serialized after variable u due to a path from $Def(v)$ to at least one $Use(u)$. To maintain correct computation, both data and control dependencies are evaluated to generate NSEs. As shown in Figure 4-a, serialization from B to C is not possible since there is a path from $Def(C)$ to $Def(E)$, which is a $Use(B)$.
- **Bidirectional NSE(u,v)** indicates that there is a NSE in both directions, (u, v) and (v, u). As shown in Figure 4-b, the live range of variable B and variable C cannot be separated by any serialization due to a bidirectional $NSE(B, C)$.
- An **NSE clique** includes a set of variables. Each pair of variables in an NSE clique has a bidirectional NSE so that all variables in the clique must be alive simultaneously. There are three NSE cliques, $\{B, C, D\}$, $\{F, G\}$ and $\{I, J\}$, shown in the example in Figure 4-c. A single variable is a degenerate case of an NSE clique.
- **Partition (E0,E1)** represents a bi-partitioning of the excessive set. As shown in Figure 4-c, the excessive set is partitioned into $E0 = \{I, J\}$, $E1 = \{B, C, D, F, G\}$.
- **Pred_set(E1)** represents a set of $Pred(Var)$ that are not in the excessive set, where Var is a variable in $E1$. As shown in Figure 5-a, $Pred\_set(E1)$=$\{A\}$.
- **Succ_set(E0)** represents a set of $Succ(Var)$ that are not in the excessive set, where Var is a variable in $E0$. As shown in Figure 5-a, $Succ\_set(E0)$=$\{K\}$.

To identify candidates for serialization, a two-step partitioning algorithm was developed to search for a partition $(E0, E1)$ which achieves the best reduction. The first step is **coarsening** where variables are merged into two partitions. To improve the partition quality, the second step, **refinement**, is applied to minimize the partition cost by moving variables between the two partitions. The partition cost is evaluated during these two phases by examining relevant cost metrics:

- **Coarsening cost metric**: This metric evaluates the number of possible variable serializations from $E0$ to $E1$. To maximize serializations, non-serializable edges (NSE) from $E0$ to $E1$ should be minimized and variables in a NSE clique should stay in the same partition. Based on this metric, the partition in Figure 4-c is feasible since there is no directed NSE from $E0$ to $E1$.
- **Refinement cost metric**: This metric evaluates whether the topology of a partition $(E0, E1)$ can lead to register reduction. Since variables in $Succ\_set(E0)$ can be simultaneously alive with $E1$ after serialization, preferably $|Succ\_set(E0)| < |E0|$. Similarly, $Pred\_set(E1)$ may be alive with $E0$ after serialization, so preferably $|Pred\_set(E1)| < |E1|$. Based on this criterion, the partitioning in Figure 5-a is a good candidate since $|Pred\_set(E1)| = 1$; $|E1| = 5$; $|Succ\_set(E0)| = 1$; $|E0| = 2$.

4.2.2 Partitioning: coarsening

The main goal of the coarsening step is to minimize the number of non-serializable edges (NSE) from $E0$ to $E1$ based on the data dependencies of the DDG. All variable pairs in the excessive set are evaluated to check whether NSE edges should be inserted. If variable u and variable v have a bidirectional NSE between them, then they should be merged into the same partition. Therefore, the first step of coarsening is to create NSE cliques.

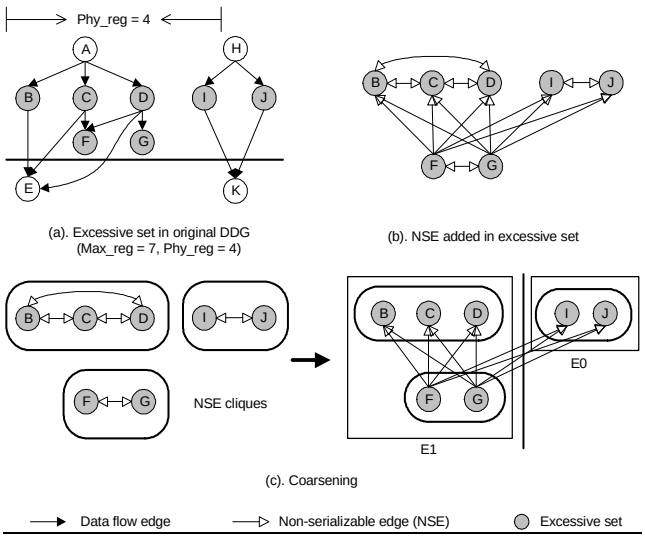

Figure 4. Partitioning: coarsening step

In general, if a set of variables is used by an operation, the variables in the set must be alive simultaneously, forming an NSE clique. Based on this rule, NSE cliques can be generated by a backward graph traversal. As shown in Figure 4-c, NSE clique $\{F, G\}$ is created first because output variables in the excessive set cannot be serialized. As a result of a backward traversal, four additional candidate NSE cliques are generated ($\{B, C, D\}$, $\{C, D\}$, $\{D\}$ and $\{I, J\}$). They are required by operations $Def(E)$, $Def(F)$, $Def(G)$ and $Def(K)$ respectively. Subsequently, two maximal NSE cliques, $\{B, C, D\}$ and $\{I, J\}$, are selected from the candidates. The pseudo code used for NSE clique generation is presented as part of Figure 6.

After NSE clique generation, the coarsening step merges two NSE cliques together based on the number of NSE edges between them. As shown in Figure 4-c, NSE clique $\{B, C, D\}$ and $\{F, G\}$ are merged together since they have the most NSE edges (6) between them. The coarsening step repetitively merges partitions until it reaches two partitions. As shown in Figure 4-c, after coarsening, the initial partition is $E0 = \{I, J\}$ and $E1 = \{B, C, D, F, G\}$. The pseudo code of the coarsening step is presented as part of Figure 6.

4.2.3 Partitioning: refinement

To improve the partition quality, the refinement step moves variables between two partitions. The partition quality is evaluated based on the partition cost, P_cost, which is defined below:

$$P\_cost = C_{reg} + C_{NSE}; \quad (1)$$

The C_{reg} term represents the non-negative gap between the maximum register requirement, Max_reg and available physical registers, Phy_reg.

$$C_{reg} = Max((Max\_reg - Phy\_reg), 0); \quad (2)$$

The Max_reg after serialization from $E0$ to $E1$ is calculated based on the topology of $E0$ and $E1$.

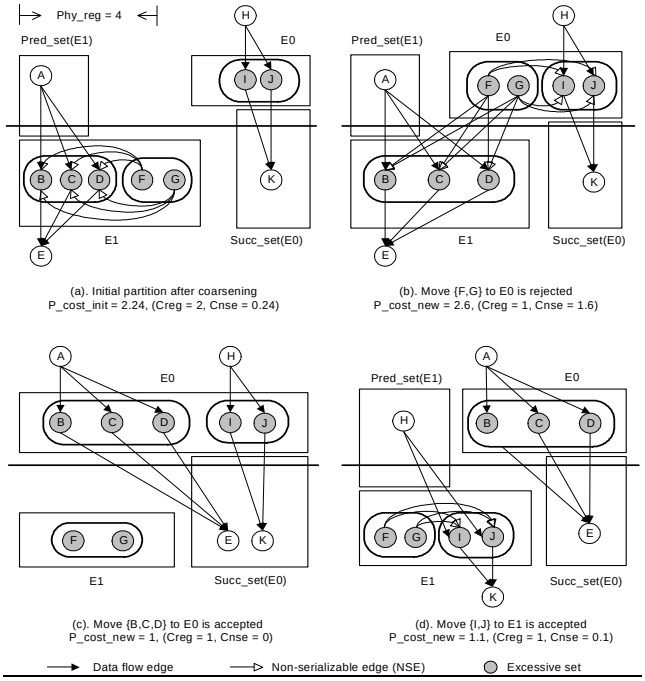

Figure 5. Partitioning: refinement step

$$Max\_reg = Max((|E0| + |Pred\_set(E1)|), (|E1| + |Succ\_set(E0)|)); \quad (3)$$

As shown in Figure 5-a, the example partition has the following topology: $|E0|$ = 2, $|Pred\_set(E1)|$ = 1; $|E1|$ = 5, $|Succ\_set(E0)|$ = 1. Assuming Phy_reg = 4, then C_{reg} = Max(3,6) - 4 = 2.

Term C_{NSE} includes two parts as shown in Equation 4. The first part, $NSE(E0, E1)$, is the number of directed non-serializable edges (NSE) from $E0$ to $E1$. The smaller the value of $NSE(E0, E1)$, the more variable serializations can be achieved from $E0$ to $E1$. To estimate the effect of non-serializable edges on the achievable register reduction, a scalar factor α = 1/Max($|E0|$, $|E1|$) is applied.

$$C_{NSE} = \alpha \times NSE(E0, E1) + \beta_0 \times NSE(E0) + \beta_1 \times NSE(E1); \quad (4)$$

The second part, $NSE(E0)$ and $NSE(E1)$, is the number of directional NSE between NSE cliques in $E0$ and $E1$, respectively. This part is only effective when there is a positive C_{reg}, which indicates another round of reduction is required to avoid spills. To allow for further reduction in the future, $NSE(E0)$ and $NSE(E1)$ should also be minimized. To estimate the effect of $NSE(E0)$ and $NSE(E1)$, scalar factor β_0 = (0.1 $\times$ C_{reg})/$|E0|$ and β_1 = (0.1 $\times$ C_{reg})/$|E1|$ are applied.

For the initial partition shown in Figure 5-a, C_{reg} = 6 - 4 = 2, C_{NSE} = $\beta_1 \times NSE(E1)$ = 0.24 and P_cost_init = C_{reg} + C_{NSE} = 2.24. To minimize the partition cost, a refinement step uncoarsens partitions and moves NSE cliques between $E0$ and $E1$. In general, if the new partition cost, P_cost_new, is smaller than the initial partition cost, P_cost_init, then the movement is accepted. The partition snapshot with the smallest P_cost is recorded and chosen as the final partition.

As shown in Figure 5-a, partition $E1$ contains two NSE cliques, $\{B, C, D\}$ and $\{F, G\}$. Partition $E0$ contains only one NSE clique, $\{I, J\}$. The refinement of this example is described below:

ES: Excessive set
Partition($E0$,$E1$): A bi-partitioning of the excessive set
NSE: Non-serializable edge
NSE clique: Variables that cannot be serialized respect to each other
P_cost: The cost of a partition $(E0, E1)$

Construct excessive set (ES)
Add non-serializable edges (NSE) between variables in excessive set

***** *NSE clique generation*
Label all variables in the excessive set as uncovered and others as covered
Put all uncovered output variables into a NSE clique, label variables as covered
Apply a backward topological traversal of the DDG
    Put uncovered variables required by an operation into a new candidate NSE clique
While there are uncovered variables
    Select the largest candidate NSE clique, label variables as covered
    Update other candidate NSE cliques and continue
EndWhile

***** *Partitioning: coarsening step. NSE cliques serve as initial partitions*
While there are more than 2 partitions
    If partitions are connected by an NSE
        Merge the two partitions connected by the most NSEs
    Else
        Merge two partitions arbitrarily
    EndIf
EndWhile
Generate a 2-way partition ($E0$,$E1$)

***** *Partitioning: refinement step*
While partitions contain more than one NSE clique
    Un-coarsen partitions and calculate initial cost, P_cost_init
    Evaluate NSE clique moves from $E0$ to $E1$, then $E1$ to $E0$
        Accept move if $P\_cost\_new < P\_cost\_init$
    Record the partition snapshot with the smallest P_cost
EndWhile
Choose the final partition ($E0$,$E1$) with the smallest P_cost

---

Figure 6. Tetris reduction: partitioning

1. Move $\{F, G\}$ from $E1$ to $E0$ as shown in Figure 5-b. After this movement, C_{reg} = 1 and C_{NSE} = $\alpha \times NSE(E0, E1) + \beta_0 \times NSE(E0)$ = 1.6. Because P_cost_new (2.6) $\geq$ P_cost_init (2.24), this movement is rejected.
2. Move $\{B, C, D\}$ from $E1$ to $E0$ as shown in Figure 5-c. This movement reduces C_{reg} to 1 and C_{NSE} to 0. Because P_cost_new (1) $<$ P_cost_init (2.24), this movement is accepted.
3. After step 2, $\{I, J\}$ is moved from E0 to E1 as shown in Figure 5-d. After this movement, C_{reg} = 1 and C_{NSE} = $\beta_1 \times NSE(E1)$ = 0.1. Because P_cost_new (1.1) $<$ P_cost_init (2.24), this movement is also accepted.

In this example, the partition in Figure 5-c has a minimum P_cost of 1. Therefore, the final partition is $E0$ = $\{B, C, D, I, J\}$ and $E1$ = $\{F, G\}$. Pseudo code for the refinement step is presented at the bottom of Figure 6.

4.3 Serialization

Serialization is applied after partitioning. The goal of this step is to select and apply serialization $(E0 \rightarrow E1)$, forming a maximal set of **compatible serializations** from E0 to E1.

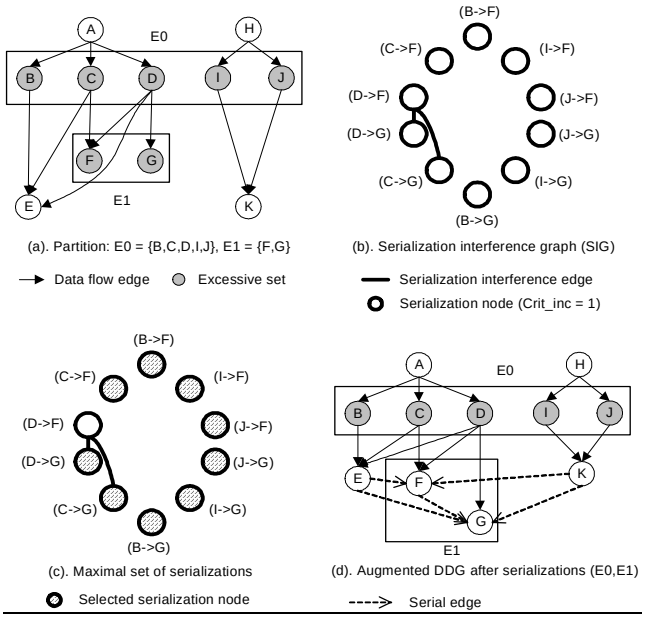

Figure 7. An example of serialization

As discussed in Section 3, **compatible serializations** represent a set of serializations that can be applied together without causing cycles. Cycles caused by incompatible serializations can inhibit any possible schedule. Figure 7 shows that serialization $(D \rightarrow F)$ requires a serial edge ($Def(G)$ to $Def(F)$) while serialization $(D \rightarrow G)$ requires a serial edge ($Def(F)$ to $Def(G)$). Applying both serializations causes a cycle between $Def(F)$ and $Def(G)$, which makes scheduling impossible. Therefore, serializations $(D \rightarrow F)$ and $(D \rightarrow G)$ are not compatible and they cannot be applied simultaneously.

In order to select a maximum set of compatible serializations from E0 to E1, a two-step serialization algorithm is used. The first step checks compatibility between serializations and creates a **serialization interference graph (SIG)** . The second step selects and applies a maximal set of serializations based on the SIG.

4.3.1 SIG construction

To represent serialization compatibility, this step creates a new graph called a serialization interference graph (SIG) based on the partitioning result $(E0, E1)$. In a SIG, each node is called a serialization node and each undirected edge is called a serialization interference edge. These terms are formally defined below:

- A **serialization node** represents a potential serialization $(u \rightarrow v)$ when there is no $NSE(u, v)$. Variable u/v is a variable in $E0$/$E1$ respectively. As shown in Figure 7-a, the partition, $E0$ = $\{B, C, D, I, J\}$ and $E1$ = $\{F, G\}$, has no $NSE(E0, E1)$. Therefore, the corresponding SIG in Figure 7-b contains all 10 potential serialization nodes from $\{B, C, D, I, J\}$ to $\{F, G\}$.
- A **serialization interference edge** between two serialization nodes indicates that the nodes are incompatible and they cannot be applied together. As shown in Figure 7-b, serialization $\{D \rightarrow F\}$ is connected to serializations $\{D \rightarrow G\}$ and $\{C \rightarrow G\}$.

A **compatibility check** evaluates all pairs of serialization nodes in a SIG. If two serializations $(u \rightarrow v)$ and $(s \rightarrow t)$ are **incompatible**, then there must be a cycle caused by both serial edges ($Use(u)$ to $Def(v)$) and ($Use(s)$ to $Def(t)$). Such cycles can only exist if there is a path from $Def(t)$ to $Use(u)$ and another path from $Def(v)$ to $Use(s)$. A path from $Def(t)$ to $Use(u)$ indicates either there is a $NSE(u, t)$ or $Def(t)$ is a $Use(u)$. Similarly, a path from $Def(v)$ to $Use(s)$ indicates either there is a $NSE(s, v)$ or $Def(v)$ is a $Use(s)$. Therefore, serializations $(u \rightarrow v)$ and $(s \rightarrow t)$ are **incompatible** if and only if at least one of following conditions is true:

1. $Def(v)$ is a $Use(s)$ and $Def(t)$ is a $Use(u)$.
2. $Def(v)$ is a $Use(s)$ and there is a $NSE(u, t)$.
3. There is a $NSE(s, v)$ and $Def(t)$ is a $Use(u)$.
4. There is a $NSE(s, v)$ and a $NSE(u, t)$.

The pseudo code of SIG construction is presented as part of Figure 8.

4.3.2 Maximal serializations

Since two compatible serialization nodes are not connected (independent) in a SIG, determining the maximum set of compatible serializations for a SIG is equivalent to finding the SIG maximum independent set. This maximum independent set problem has previously been shown to be NP-complete [7]. To address this issue, we have developed a heuristic to find the maximal set of serializations. Our heuristic uses a serialization cost function S_cost that includes two terms, N_deg and $Crit\_inc$.

$$S\_cost = \gamma \times N\_deg + Crit\_inc; \quad (5)$$

The N_deg term is the SIG node degree, the number of serialization interference edges connected to the node. As shown in Figure 7-b, serialization node $\{D \rightarrow F\}$ has a node degree of 2, which indicates that it is not compatible with two other serializations.

$Crit\_inc$ represents the estimated increase of the critical path. $Crit\_inc$ caused by a serialization $(u \rightarrow v)$ is defined as:

$$Crit\_inc = Max((ltime(Use(u)) - ltime(Def(v))), 0); \quad (6)$$

where $ltime$ represents the latest time a node can be scheduled without increasing the DDG critical path. The latest time of an operation can be calculated by a backward graph traversal using the following equation:

$$ltime(Def(v)) = Min(ltime(Use(v)) - Delay(Def(v))); \quad (7)$$

As shown in Figure 7-a, the critical path of the DDG is A-B-E. $Def(A)$ and $Def(H)$ have an $ltime$ of 1. $Def(B)$, $Def(C)$, $Def(D)$, $Def(I)$ and $Def(J)$ have an $ltime$ of 2. $Def(E)$, $Def(F)$, $Def(G)$ and $Def(K)$ have an $ltime$ of 3. For serialization $(B \rightarrow F)$, a serial edge ($Def(E)$ to $Def(F)$) is required, which increases the critical path by 1, from A-B-E to A-B-E-F. For the SIG shown in Figure 7-b, all serialization nodes have the same $Crit\_inc$ of 1.

To maximize the total number of compatible serializations, the scalar factor γ is set to 1024 so that the serialization node with the smallest N_deg is always selected first. To control the critical path increase caused by serial edges, a threshold is set to prevent certain serializations. In our experiments, if the $Crit\_inc$ of a serialization node is larger than 3 times of the latency of a load operation, it is not applied.

The SIG in Figure 7-c illustrates that our heuristic continuously selects the serialization node with the smallest S_cost until there are no more compatible serialization nodes left in the SIG. When a serialization node $(u \rightarrow v)$ is selected, serial edges ($Use(u)$ to

Partition($E0$,$E1$): A bi-partitioning of the excessive set
SIG: Serialization interference graph
MCS: Maximal set of compatible serialization nodes in a SIG
S_cost: Cost of a serialization node
N_deg: Node degree of a serialization node
Crit_inc: Estimated critical path increase caused by a serialization

***** *Construct a SIG based on partition($E0$,$E1$)*
For each serialization $(u \rightarrow v)$, where u/v is a variable in $E0$/$E1$
    Apply NSE check
    Add a serialization node in the SIG if there is no NSE(u,v)
EndFor
For each pair of serialization nodes in the SIG
    Apply compatibility check
    Add an edge between two nodes if they are incompatible
EndFor
For each serialization node in the SIG
    Calculate N_deg and $Crit\_inc$
    Assign a S_cost based on N_deg and $Crit\_inc$
EndFor

***** *Select a maximal set of compatible serialization nodes*
Initialize an empty MCS
While there are serialization nodes that are independent of MCS
    Add a serialization $(u \rightarrow v)$ with the minimum S_cost into MCS
    Add serial edges ($Use(u)$ to $Def(v)$) to the DDG
EndWhile

Figure 8. Tetris reduction: serialization

$Def(v)$) are inserted into the DDG. The pseudo code of this step is presented as part of Figure 8.

For the example in Figure 7-c, all serialization nodes have the same $Crit\_inc$ of 1. The serialization node with the minimum N_deg is selected and applied first. The serialization process is shown below:

1. Select 7 serialization nodes with N_deg of 0:

- $\{C \rightarrow F\}$ requires a serial edge ($Def(E)$ to $Def(F)$).
- $\{B \rightarrow F\}$ requires a serial edge ($Def(E)$ to $Def(F)$).
- $\{I \rightarrow G\}$ requires a serial edge ($Def(K)$ to $Def(G)$).
- $\{J \rightarrow G\}$ requires a serial edge ($Def(K)$ to $Def(G)$).
- $\{I \rightarrow F\}$ requires a serial edge ($Def(K)$ to $Def(F)$).
- $\{J \rightarrow F\}$ requires a serial edge ($Def(K)$ to $Def(F)$).
- $\{B \rightarrow G\}$ requires a serial edge ($Def(E)$ to $Def(G)$).

2. Select 2 serialization nodes with N_deg of 1:

- $\{D \rightarrow G\}$ requires 2 serial edges ($Def(E)$ to $Def(G)$) and ($Def(F)$ to $Def(G)$).
- $\{C \rightarrow G\}$ requires 2 serial edges ($Def(E)$ to $Def(G)$) and ($Def(F)$ to $Def(G)$).

After step 2, there is no compatible serialization node left in the SIG since $\{D \rightarrow F\}$ interferes with both $\{D \rightarrow G\}$ and $\{C \rightarrow G\}$.

As shown in Figure 7-d, after applying the above 9 serializations, the augmented DDG contains 5 serial edges, ($Def(E)$ to $Def(F)$), ($Def(K)$ to $Def(G)$), ($Def(K)$ to $Def(F)$), ($Def(E)$ to $Def(G)$) and ($Def(F)$ to $Def(G)$). Max_reg of the augmented DDG is reduced from 7 to 5. The new excessive set is $\{B, C, D, I, J\}$.

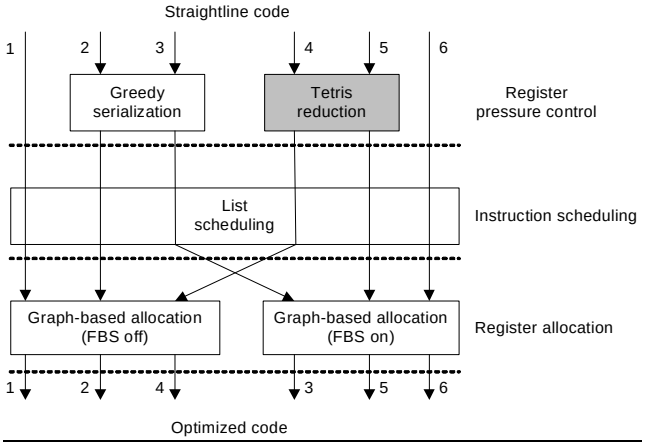

Figure 9. Experimental flow in Trimaran framework

5. Experimental approach and results

To evaluate the effectiveness of our new reduction algorithm, a direct comparison to several pre-existing spill code reduction techniques was performed. These reduction techniques (including Tetris) were implemented in an academic VLIW compiler, Trimaran [6], which includes a front-end, a backend and a simulator. Tetris reduction is integrated into the backend, ELCOR. Trimaran allows users to modify the number of target functional units (FUs), registers and other resources to allow for examination of a broad range of VLIW architectures. Benchmarks in our experiments include several programs taken from the Trimaran framework [6] and three applications taken from the MediaBench suite [14]. Benchmarks *unepic*, *g721dec* and *mpeg2dec* are applications for image, audio and video signal processing, respectively.

As shown in Figure 9, our experimental flow includes a register pressure control step, a scheduling step and a register allocation step. An existing register pressure control algorithm, the greedy serialization technique [19] discussed in Section 3, was implemented for comparison with our new algorithm. After register pressure control, the default list scheduling algorithm in the Trimaran framework is applied, followed by graph-based register allocation with FBS turned on or off. FBS is the frequency-based live range splitting technique [13] described in Section 2.

To evaluate the performance of each individual technique (Tetris reduction, greedy serialization and FBS), we first compare flow 1 to flows 2, 4, and 6. Flow 1 is the baseline Trimaran flow without register pressure control or FBS. Flow 2 and flow 4 apply greedy serialization and Tetris reduction, respectively, with FBS off. Flow 6 applies FBS with no register pressure control.

The first VLIW architecture evaluated in our experiments is a 4-way VLIW architecture with 16 registers, which can execute 4 operations (including 2 memory operations) on every clock cycle. This resource configuration can be found in several low-end commercial VLIW processors including the Freescale MSC8101 and MSC8103 [9]. These processors are often used in resource-constrained embedded systems. Our first experiment evaluates the benefit of each individual technique (Tetris reduction, greedy serialization and FBS) for the 4-way architecture. Results for each flow are shown in 3 separate columns in Table 1. In the first column, benchmark clock cycles indicate application performance. The second column shows the number of dynamic spill operations for each benchmark. Both clock cycles and dynamic spill operations are shown in thousands of values. In the third column, spill ratio, the percentage of dynamic spill operations to the total number of operations, is presented. A high spill ratio indicates that a benchmark

Benchmarks	Tetris (flow 4)			Greedy (flow 2)			FBS (flow 6)			Base (flow 1)		
	Cycles (K)	Spills (K)	Spill%	Cycles (K)	Spills (K)	Spill%	Cycles (K)	Spills (K)	Spill%	Cycles (K)	Spills (K)	Spill%
bmm	325.64	195.74	0.594	325.64	195.74	0.594	314.36	189.44	0.586	325.64	195.74	0.594
mm	215.86	143.09	0.531	292.26	178.04	0.568	330.54	182.58	0.609	300.61	174.99	0.562
mm_double	290.66	168.07	0.553	290.65	168.07	0.553	319.55	171.53	0.585	290.65	168.07	0.553
mm_dyn	190.50	123.88	0.494	332.79	238.32	0.648	358.34	252.53	0.655	348.68	252.88	0.656
parms_test	12.37	6.84	0.435	12.37	6.84	0.434	13.28	7.84	0.468	12.37	6.84	0.435
sqrt	4.68	2.24	0.411	7.39	4.09	0.546	8.79	4.38	0.562	7.96	4.00	0.540
strcpy	25.47	7.77	0.283	26.49	8.54	0.302	27.14	8.62	0.371	33.06	16.01	0.448
switch_test	13.01	2.58	0.141	13.01	2.58	0.141	13.01	2.58	0.141	13.01	2.58	0.141
wave	34.51	19.30	0.441	50.27	29.78	0.541	44.84	29.79	0.539	60.43	35.49	0.582
unepic	13715	7715	0.400	14992	8892	0.434	18535	11153	0.478	17944	11013	0.474
g721dec	247411	116112	0.235	270268	143463	0.270	227010	135469	0.255	300607	169149	0.298
mpeg2dec	137828	160276	0.463	216025	244131	0.559	313154	244822	0.549	286892	279593	0.583
Geomean	321.19	167.77	38%	390.98	211.44	43%	418.63	219.64	45%	429.37	236.41	46%
% change	-25%	-30%		-9%	-11%		-3%	-7%				

Table 1. Comparison of spill reduction techniques for architecture with 16 registers and 4 functional units

Benchmarks	Tetris (flow 4)			Greedy (flow 2)			FBS (flow 6)			Base (flow 1)		
	Cycles (K)	Spills (K)	Spill%	Cycles (K)	Spills (K)	Spill%	Cycles (K)	Spills (K)	Spill%	Cycles (K)	Spills (K)	Spill%
bmm	104.49	28.78	0.177	104.49	28.79	0.177	101.76	27.83	0.172	104.49	28.79	0.177
mm	108.19	20.61	0.135	114.84	44.04	0.245	114.48	40.83	0.231	114.64	40.85	0.231
mm_double	101.53	26.26	0.162	101.53	26.27	0.162	101.42	26.27	0.162	101.53	26.27	0.162
mm_dyn	103.02	30.28	0.192	107.25	77.11	0.371	98.81	57.67	0.303	125.93	81.10	0.379
parms_test	6.12	2.31	0.207	6.12	2.31	0.206	6.07	2.28	0.204	6.12	2.32	0.207
sqrt	3.99	0.22	0.065	4.17	0.73	0.177	4.17	0.67	0.163	4.20	1.01	0.229
strcpy	18.78	0.04	0.002	18.27	0.05	0.003	18.28	0.06	0.003	18.28	0.07	0.003
switch_test	12.99	2.57	0.140	12.99	2.58	0.140	12.99	2.58	0.140	12.99	2.58	0.140
wave	13.67	1.39	0.053	21.79	11.06	0.304	21.59	12.30	0.326	42.54	21.43	0.457
unepic	7553	2016	0.148	8871	3362	0.224	8784	370	0.233	9544	417	0.255
g721dec	175845	32610	0.080	175596	39313	0.224	162451	43018	0.098	176087	47488	0.107
mpeg2dec	77626	5367	0.027	82948	34726	0.149	97939	27178	0.120	108624	72519	0.265
Geomean	177.70	21.94	8%	190.11	41.92	14%	189.15	41.06	13%	209.78	50.85	15%
% change	-15%	-57%		-9%	-18%		-9%	-19%				

Table 2. Comparison of spill reduction techniques for architecture with 32 registers and 8 functional units

Benchmarks	Tetris+FBS (flow 5)			Greedy+FBS (flow 3)		
	Cycles (K)	Spills (K)	Spill%	Cycles (K)	Spills (K)	Spill%
bmm	101.75	27.83	0.172	101.757	27.83	0.172
mm	108.32	20.61	0.134	112.602	37.14	0.215
mm_double	101.41	26.26	0.162	101.416	26.27	0.162
mm_dyn	97.65	19.87	0.135	90.708	56.22	0.300
parms_test	6.07	2.27	0.204	6.070	2.27	0.204
sqrt	3.97	0.16	0.049	4.080	0.71	0.174
strcpy	18.64	0.04	0.002	18.270	0.05	0.003
switch_test	12.99	2.57	0.140	12.998	2.58	0.140
wave	13.01	1.34	0.052	19.881	11.13	0.306
unepic	8710	2907	0.200	9093	3402	0.226
g721dec	171727	28719	0.071	174243	38523	0.092
mpeg2dec	77226	4892	0.025	96919	19952	0.092
Geomean	177.22	20.76	7%	187.54	38.20	13%
% change	-15%	-59%		-10%	-25%	

Table 3. Comparison of combined spill reduction techniques for architecture with 32 registers and 8 functional units

Benchmarks	Tetris (flow 4)			Greedy (flow 2)			FBS (flow 6)			Base (flow 1)		
	Cycles (K)	Spills (K)	Spill%	Cycles (K)	Spills (K)	Spill%	Cycles (K)	Spills (K)	Spill%	Cycles (K)	Spills (K)	Spill%
bmm	87.50	0.28	0.002	87.51	0.28	0.002	87.56	0.28	0.002	87.51	0.28	0.002
mm	93.72	1.67	0.013	92.63	1.75	0.013	92.59	1.75	0.013	92.59	1.75	0.013
mm_double	89.34	1.70	0.012	89.34	1.71	0.012	89.34	1.71	0.012	89.34	1.71	0.012
mm_dyn	29.22	1.68	0.013	29.93	1.72	0.013	29.88	1.73	0.013	29.89	1.73	0.013
parms_test	5.61	1.76	0.165	5.62	1.76	0.165	5.62	1.74	0.164	5.62	1.76	0.165
sqrt	3.84	0.13	0.038	3.75	0.15	0.041	3.77	0.19	0.052	3.77	0.17	0.052
strcpy	19.03	0.04	0.002	18.27	0.05	0.002	18.27	0.05	0.002	18.27	0.05	0.002
switch_test	12.99	2.57	0.140	12.99	2.58	0.140	12.99	2.58	0.140	12.99	2.58	0.140
wave	12.94	0.05	0.002	12.14	0.10	0.004	12.09	0.13	0.005	12.09	0.13	0.005
unepic	5880	55	0.005	7236	1632	0.122	6613	1443	0.106	7126	1737	0.125
g721dec	162225	30840	0.076	165303	40920	0.097	162189	41717	0.095	162786	42896	0.098
mpeg2dec	75318	4043	0.021	74557	5145	0.025	74590	5559	0.027	74673	5680	0.028
Geomean	147.52	3.87	1.4%	148.76	5.86	2.1%	147.61	6.07	2.2%	148.39	6.20	2.3%
% change	-1%	-37%		0	-6%		-1%	-2%				

Table 4. Comparison of spill reduction techniques for architecture with 64 registers and 8 functional units

suffers from high register pressure. At the bottom of the table, the geometric average of values is provided along with the percentage change versus the baseline Trimaran flow.

For the baseline (flow 1) in Table 1, on average, spills take up 46% of total executed operations. As shown in Table 1, Tetris reduction (flow 4), greedy serialization (flow 2) and FBS (flow 6) reduce spills by 30%, 11% and 7%, respectively. Due to these spill reductions, the average execution cycles of benchmarks are reduced by 25%, 9% and 3%. For this 4-way VLIW architecture, Tetris reduction outperforms both greedy serialization and FBS in terms of spill reduction and performance improvement. The benefit of Tetris reduction is a result of its ability to reduce register pressure. Compared with FBS, Tetris reduction reduces the maximum register requirement by 18%, which helps register allocation reduce the number of spills. Greedy serialization also reduces register pressure. However, unlike Tetris reduction, which selects and serializes multiple variables simultaneously, greedy serialization only performs a single variable serialization at a time. The greedy approach prevents simultaneous variable serializations and limits the total serialization benefit. Compared with greedy serialization, Tetris reduction allows 75% more serial edges to be inserted into a DDG on average versus Greedy, which provides an additional 9% reduction in the maximum register requirement. Reduced register pressure helps register allocation reduce spills by an additional 19% and improves performance by 16% for Tetris versus the greedy approach.

The second VLIW architecture evaluated in our experiments is an 8-way architecture with 32 registers. This architecture can execute 8 operations (including 2 memory operations) on every clock cycle. This configuration is similar to the VLIW processors found in the C62x and C67x families offered by Texas Instruments [17]. For a register size of 32, the baseline (flow 1) in Table 2 has a spill ratio of 15%. Compared with the baseline flow, the average spill reduction achieved by Tetris reduction (flow 4), greedy serialization (flow 2) and FBS (flow 6) is 57%, 18% and 19%, respectively. This reduction improves performance by 15%, 9% and 9%. Compared with greedy serialization, Tetris reduction achieves an additional spill reduction of 39% because it provides an additional 8% reduction in the maximum register requirement.

For some designs, Tetris achieves slightly reduced performance versus other approaches. For example, as shown in Table 2, benchmark *strcpy*, is adversely affected by the use of Tetris reduction. Although Tetris reduces the baseline case spill ratio from 0.3% to 0.2%, the benefit of spill reduction is outweighed by the critical path increase caused by serial edges. As a result, performance achieved by Tetris versus the baseline case is degraded by 2.7%. In future work, it may be possible to modify the serialization cost so that fewer serial edges are inserted when the spill ratio is low.

As an additional experiment, we evaluate the use of register pressure control and FBS together in the same flow. Table 3 shows the performance of Tetris reduction with FBS (flow 5) and greedy serialization with FBS (flow 3). On average, Tetris with FBS reduces spills by 59%, and improves performance by 15% versus the baseline. This result is 5% better than the performance speedup offered by greedy serialization with FBS.

In summary, when register pressure is relatively high (average spill ratio of 15%), Tetris reduction outperforms other techniques by at least 6% in terms of cycle count.

An 8-way VLIW machine with 64 registers was used for a final experiment. This architecture has the same basic FU and register configuration as the Transmeta Efficeon VLIW processor [20] and the Texas Instruments C64x processor [17]. On average, spills take up 2.3% of the total operations in the baseline flow in Table 4. Due to this low register pressure, Tetris reduction (flow 4) provides a performance speedup of 1% and spills are reduced by 37%. As expected, the benefit of Tetris reduction becomes marginal when register pressure is very low.

6. Summary

In this paper, we present a new technique to improve the performance for VLIW processors by reducing register pressure. Our Tetris reduction modifies the relative ordering of operations to serialize multiple variables simultaneously so that the maximum register requirement is reduced. This technique reduces spills and improves execution time for VLIW programs that experience high register pressure. Compared with previous work, the execution time is reduced on average by 16% for a 4-way VLIW architecture with 16 registers and 6% for a 8-way VLIW architecture with 32 registers. A limitation of the current Tetris technique is that it may cause performance degradation when the register pressure of an application is very low. This issue will be further evaluated in future work.

Acknowledgments

This work was sponsored by National Science Foundation grant CCR-9988238. The authors wish to acknowledge the efforts of Premachandran R. Menon in the completion of this work.

References

[1] D. A. Berson, R. Gupta, and M. L. Soffa. URSA: A Unified ReSource Allocator for Registers and Functional Units in VLIW Architectures. In *IFIP Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, pages 243–254, Jan. 1993.

[2] D. A. Berson, R. Gupta, and M. L. Soffa. Integrated Instruction Scheduling and Register Allocation Techniques. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 247–262, Aug. 1998.

[3] P. Briggs. *Register Allocation via Graph Coloring*. PhD thesis, Department of Computer Science, Rice University, Apr. 1992.

[4] P. Briggs, K. Cooper, K. Kennedy, and L. Torczon. Coloring Heuristics for Register Allocation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 275–284, June 1989.

[5] G. Chaitin. Register Allocation and Spilling via Graph Coloring. In *ACM SIGPLAN Symposium on Compiler Construction*, pages 98–105, June 1982.

[6] L. N. Chakrapani, J. Gyllenhaal, W. W. Hwu, S. A. Mahlke, K. V. Palem, and R. M. Rabbah. Trimaran, An Infrastructure for Research in Instruction Level Parallelism. In *International Workshop on Languages and Compilers for High Performance Computing*, pages 32–41, Sept. 2004.

[7] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. McGraw-Hill Book Company, 1990.

[8] R. P. Dilworth. A Decomposition Theorem for Partially Ordered Sets. *Annals of Mathematics*, 51(1):161–166, Jan. 1950.

[9] Freescale Semiconductor, Inc. *MSC8101 Reference Manual*, 2005.

[10] S. M. Freudenberger and J. C. Ruttenberg. Phase Ordering of Register Allocation and Instruction Scheduling. In *International Workshop on Code Generation*, pages 146–172, May 1991.

[11] J. R. Goodman and W.-C. Hsu. Code scheduling and register allocation in large basic blocks. In *ACM Supercomputing Conference*, pages 442–452, July 1988.

[12] R. Govindarajan, H. Yang, J. N. Amaral, C. Zhang, and G. R. Gao. Minimum Register Instruction Sequencing to Reduce Register Spills in Out-of-Order Issue Superscalar Architectures. *IEEE Transactions on Computers*, 52(1):4–20, Jan. 2003.

[13] H. Kim. *Region-based Register Allocation for EPIC Architectures*. PhD thesis, Department of Computer Science, New York University, Jan. 2001.

[14] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communicatons Systems. In *International Symposium on Microarchitecture*, pages 330–335, June 1997.

[15] C. Norris and L. L. Pollock. A Scheduler-Sensitive Global Register Allocator. In *ACM Supercomputing Conference*, pages 804–813, July 1993.

[16] S. S. Pinter. Register Allocation with Instruction Scheduling: A New Approach. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 248–257, June 1993.

[17] Texas Instruments, Inc. *TMS320C6000 CPU and Instruction Set Reference Guide*, 2000.

[18] S.-A.-A. Touati. Register Saturation in Superscalar and VLIW Codes. In *International Conference on Compiler Construction*, pages 213–228, Apr. 2001.

[19] S.-A.-A. Touati. Register Saturation in Instruction Level Parallelism. *International Journal of Parallel Programming*, 33(4):393–449, Aug. 2005.

[20] Transmeta, Inc. *Transmeta Efficeon TM8820 Processor*, 2005.