Tetris-XL:A Performance-Driven Spill Reduction Technique for Embedded VLIW Processors

Weifeng Xu and Russell Tessier
University of Massachusetts Amherst

As technology has advanced, the application space of Very Long Instruction Word (VLIW) processors has grown to include a variety of embedded platforms. Due to cost and power consumption constraints, many embedded VLIW processors contain limited resources, including registers. As a result, a VLIW compiler that maximizes instruction level parallelism (ILP) without considering register constraints may generate excessive register spills, leading to reduced overall system performance. To address this issue, this paper presents a new spill reduction technique that improves VLIW runtime performance by reordering operations prior to register allocation and instruction scheduling. Unlike earlier algorithms, our approach explicitly considers both register reduction and data dependency in performing operation reordering. Data dependency control limits unexpected schedule length increases during subsequent instruction scheduling. Our technique has been evaluated using Trimaran, an academic VLIW compiler, and evaluated using a set of embedded systems benchmarks. Experimental results show that, on average, this technique improves VLIW performance by 10% for VLIW processors with 32 registers and 8 functional units compared with previous spill reduction techniques. Limited improvement is seen versus prior approaches for VLIW processors with 64 registers and 8 functional units.

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

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Register Pressure, Instruction Level Parallelism, Very Long Instruction Word (VLIW) Processor

1. INTRODUCTION

VLIW processors are currently used in a variety of embedded systems that require high performance within constrained operating conditions [Goossens et al. 1997; Faraboschi et al. 2000]. In an effort to minimize hardware, typical VLIW processors do not provide specialized hardware to support dynamic scheduling or out of order execution [Hennessy and Patterson 1996]. Rather, compile-time scheduling is used to determine a fixed schedule for multiple operations performed in parallel on VLIW functional units. As a result, the runtime performance of a VLIW processor depends

A preliminary version of this paper was presented at ACM SIGPLAN LCTES 2007, San Diego, CA, in June 2007. This work was supported by National Science Foundation grant CCR-9988238.
Authors' address: Department of Electrical and Computer Engineering, University of Massachusetts, Amherst, MA 01003.

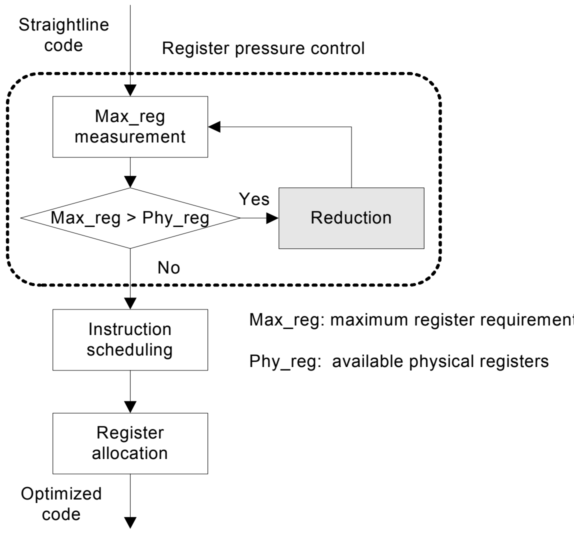

Fig. 1. VLIW compilation flow with register pressure control

heavily on the efficiency of its compiler.

A typical VLIW compilation flow includes several optimization phases, including instruction scheduling and register allocation [Freudenberger and Ruttenberg 1991]. Instruction scheduling attempts to maximize ILP by scheduling as many operations 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 that is used. Given the latencies involved in main memory access, spills may result in significantly reduced system performance.

Due to cost and power consumption constraints of embedded systems, many embedded VLIW processors only contain a limited number of physical registers, exacerbating the possibility of memory spills. For example, the Freescale MSC8101 [Freescale Semiconductor, Inc. 2005] and TI C62x [Texas Instruments, Inc. 2000] VLIW processors contain 16 and 32 physical registers, respectively. In these situations, if a VLIW compiler simply maximizes ILP without considering register constraints, a substantial number of spills may be generated. However, simply minimizing spills may not always improve performance if application parallelism is negatively affected. In this paper we show that both of these issues must be taken into account during the early stages of compilation to achieve the best possible application performance.

A number of spill reduction techniques [Kim 2001; Govindarajan et al. 2003; Touati 2005], including register pressure control [Touati 2005], have been developed. As shown in Figure 1, register pressure control is 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 technique based on the reordering of operations. Our technique reorders operations to reduce the number of required registers and spills while attempting to maintain instruction level parallelism. To demonstrate its benefit, we have integrated this algorithm into an academic VLIW compiler, Trimaran [Chakrapani et al. 2004]. As shown in Section 6, the algorithm can achieve at least a 10% reduction in benchmark execution time compared with previously-published approaches [Kim 2001; Govindarajan et al. 2003; Touati 2005] for a VLIW architecture with 32 registers and 8 functional units at the cost of a modest compile time increase. For 64 register VLIW architectures with lower register pressure, all approaches perform roughly equally.

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. Section 4 presents a heuristic called Tetris that focuses on reducing the maximum register requirement (Max_reg) without considering data dependencies. In Section 5, an extension of Tetris, called Tetris-XL, is presented. This algorithm considers both register reduction and data dependencies. Our experimental approach and results are presented in Section 6. Section 7 provides a summary of this work.

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, integrated register allocation and 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 [Chaitin 1982; Briggs et al. 1989; Briggs 1992; Bouchez et al. 2007] 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 [Kim 2001] 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. A limitation of general register allocation is that once an instruction schedule has been determined, the flexibility needed to avoid register overuse is significantly reduced, regardless of the allocation algorithm. Our approach attempts to predict register overusage before instruction scheduling to make subsequent register allocation more effective.

Schedule-sensitive register allocation techniques [Norris and Pollock 1993; Pinter 1993] 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) [Govindarajan et al. 2003]. MRIS applies a special interference graph, where each node represents an instruction lineage, a series of variables that can share the same physical register. If a lineage interference graph requires more registers than available physical registers, a fusion step is applied to reorder operations so that two lineages can share the same register. Based on the coloring result of the lineage interference graph, MRIS applies a modified list scheduling algorithm, which schedules operations based on the availability of physical registers. A limitation of this approach is the insertion of additional name dependency created by assigning registers before scheduling. As a result, the achievable schedule length may be increased, causing performance degradation. Unlike schedule-sensitive register allocation, register pressure control does not assign registers so it avoids the adverse effect of name dependency. In Section 6 it is shown that our register pressure control technique outperforms schedule-sensitive register allocation techniques, such as MRIS.

Register-sensitive instruction scheduling controls register requirements during instruction scheduling, which is performed before register allocation. Goodman and Hsu [Goodman and Hsu 1988] 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. Register-sensitive instruction scheduling is limited since scheduling heuristic switching is based on the instantaneous register requirement. Because the register requirement after switching is not predicted, the scheduler may not be able to reduce register usage and avoid spills. Our new approach addresses register pressure earlier in the compilation flow to allow for more efficient exploration of the reduced register usage space.

Integrated register allocation and instruction scheduling techniques attempt to ensure register constraints while optimizing performance during scheduling. In the Register on Demand (RoD) algorithm [Cilio and Corporaal 1999], an operation is scheduled only if a free register and functional unit are present. Register assignment or spill insertion are done on the fly. Because spill insertion is based on the instantaneous register and schedule requirements, it may be difficult to consider the effect of operation interdependencies. A second integrated technique [Zeitlhofer and Wess 2003] ensures that only schedules that satisfy all register constraints are generated. As a result, final register assignment is guaranteed to be successful. Our new approach attempts to address performance by considering an assessment of operation orderings while leaving specific register allocation and scheduling to later in the compilation flow.

Unlike the above approaches, register pressure control has more freedom in moving operations to reduce spills. Berson presented a technique called unified resource allocation (URSA) to reduce register pressure so that the schedule generated in

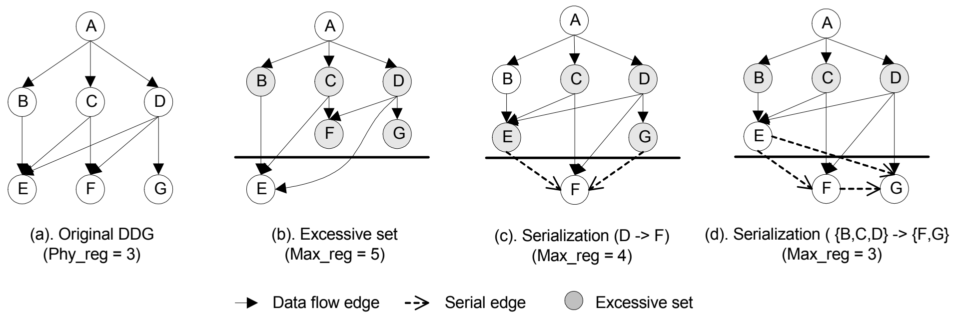

Fig. 2. Register pressure reduction via variable serializations

subsequent instruction scheduling does not overuse registers [Berson et al. 1993; 1998]. This measure-and-reduce methodology is shown in Figure 1. Experimental results [Berson et al. 1998] show that register pressure control outperforms both schedule-sensitive register allocation (PIG) and register-sensitive instruction scheduling (IPS). Based on Berson's measure-and-reduce approach, Touati presented several new heuristics to further improve register pressure control [Touati 2001; 2005]. These extensions are discussed in Section 3. In general, previous register pressure control approaches only consider the movement of an individual operation at a time. In situations with high ILP, it is more beneficial to consider groups rather than individual operations [Berson et al. 1998]. To address this limitation, our heuristics consider the effect of moving a set of operations together, which allows for a better register reduction.

The work described in this paper extends an earlier version of our previous register pressure control algorithm [Xu and Tessier 2007] in several important ways. As described in [Xu and Tessier 2007], our earlier algorithm only considers register reduction when performing operation reordering. Although this approach aggressively reduces required register count and associated spills, generated data dependencies may lead to reduced instruction level parallelism. This issue may lead to increased application cycle counts as a result of longer instruction schedules. This paper also provides a comparison of our register pressure control algorithms to results generated by MRIS [Govindarajan et al. 2003], an early register allocation technique. This comparison was missing from our earlier work.

3. BACKGROUND

In this section, we first present several basic definitions. After discussing the limitation of previous techniques [Touati 2001; 2005], 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), $G(V, E)$. A DDG contains a set of nodes V and a set of directed edges $E = u, v : u, v \in E$. A node $u \in V$ represents an operation that defines variable u. A directed edge $(u, v) \in E$ represents a data flow, where node v uses the variable defined by node u.

The delay of node u is equal to $d(u)$ clock cycles. Node u reads registers on the first cycle of $d(u)$ and writes the register on the last cycle. Our heuristics consider both unit and multi-cycle delay operations. However, for demonstration purposes, subsequent examples shown in the figures assume all nodes have unit delay, $d(u) = 1$. Additional terms are defined as follows:

—**Pred(u)** $= v \in V : (v, u) \in E$ is a set of predecessor nodes required by u. Figure 2-a shows that $Pred(E) = \{B, C, D\}$.

—**Succ(u)** $= v \in V : (u, v) \in E$ is a set of successor nodes that use u as an input. Figure 2-a shows that $Succ(A) = \{B, C, D\}$.

—A **Use(u)** is node $v \in V$ such that $v \in Succ(u)$. A node v which is a $Use(u)$ uses variable u as an input.

—**Lv(u)** is the *live range* of variable u. $Lv(u)$ is the distance from node u to the last $Use(u)$. As shown in Figure 2-a, live range $Lv(B)$, is from node B to node E, the only node which uses variable B. Live ranges are independent of original statement ordering and are only dependent on variable definition and use nodes.

—An **excessive set** (ES) is a maximal set of nodes (variables) in a DDG which can be alive simultaneously such that the size of the set, Max_reg, exceeds the number of available physical registers, Phy_reg. Formally, ES is the maximal set of nodes $v \in V$ that satisfies the following conditions: (a) $\forall u, v \in ES, Lv(u) \cap Lv(v) \neq \emptyset$; (b) $|ES| > Phy\_reg$. Multiple excessive sets may exist for a given DDG.

For a given DDG, the maximum register requirement is the largest number of variables that are 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 [Berson et al. 1993], 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 [Dilworth 1950]. In this paper, we use an improved estimation technique called register saturation [Touati 2005]. Previous results show that the estimated result is within one register of the measured maximum register requirement [Touati 2005].

Using register saturation [Touati 2005], the maximum register requirement of the DDG in Figure 2-a is 5. As shown in Figure 2-b, if E is scheduled last, 5 variables $\{B, C, D, F, G\}$ are alive simultaneously since variables $\{B, C, D\}$ required by node E and $\{F, G\}$ are output variables. If there are less than 5 physical registers, $\{B, C, D, F, G\}$ becomes an excessive set, ES.

To reduce the size of the excessive set, live ranges of variables in the excessive set must be separated so that they do not overlap. 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:

—A **serial edge** (w **to** v) is a directed edge from w to v. Serial edge (w to v) enforces an ordering such that variable v cannot be written before all inputs to node w have been read.

—**Serialization** ($u \rightarrow v$) enforces an ordering such that $Lv(v)$ begins after $Lv(u)$ ends. Formally, serialization creates serial edges (w to v): $\forall w \in (Succ(u) - v)$.

It has been proven [Touati 2005] that minimizing the excessive set size via serialization is NP-hard. To address this problem, Touati [Touati 2005] presented a greedy serialization technique, which evaluates all possible serializations between any pair of variables and selects the one which can best reduce the maximum 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 F cannot be scheduled earlier than D, two serial edges (E to F) and (G to 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 F and $\{E, G\}$ are output variables. Due to serial edges (E to F) and (G to F), the critical path changes from A-B-E to A-B-E-F.

A limitation of greedy serialization is that only a single serialization is considered by the algorithm at a time, limiting tradeoffs across multiple potential serializations. This greedy behavior often leads to poor performance. Figure 2-c shows that serialization $(D \rightarrow F)$ requires a serial edge (G to F). Additionally, serialization $(D \rightarrow G)$ requires a serial edge (F to G). Applying both serializations causes a cycle between F and G, which makes scheduling impossible. Therefore, 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:

—$AG(V, E, SE)$ is an augmented version of graph $G(V, E)$, which includes serial edges SE. To allow for a feasible scheduling, $AG(V, E, SE)$ must contain no cycles.

—Two serializations are compatible if they can be applied together without creating a cycle in $AG(V, E, SE)$. The expression $(u \rightarrow v) \mid (s \rightarrow t)$ indicates that serializations $(u \rightarrow v)$ and $(s \rightarrow t)$ are compatible.

—The expression $(set1 \rightarrow set2)$ indicates the maximal set of compatible serializations: $(u \rightarrow v) : u \in set1, v \in set2$.

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 maximal set in the augmented DDG is $\{B, C, D\}$, in which all variables are required by 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

Tetris reduction was first presented in [Xu and Tessier 2007]. The basic idea of Tetris reduction originates from the popular computer puzzle game. In a Tetris

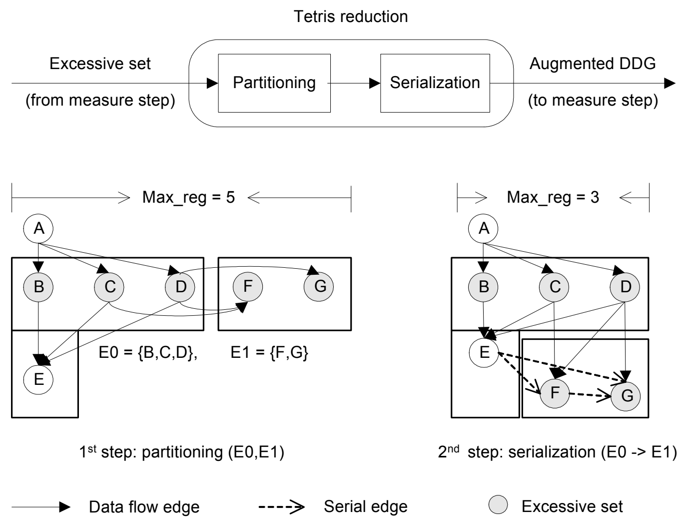

Fig. 3. Tetris: a heuristic reduction technique

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.

Each code block may be subjected to multiple iterations of the Tetris algorithm until further improvement across all excessive sets is impossible.

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. This edge indicates that serialization ($u \rightarrow v$) cannot be applied due to a path from v to at least one node which is a $Use(u)$. To maintain correct computation, both data and control dependencies are evaluated to generate NSEs. As shown in Figure 6-a, serialization ($B \rightarrow C$) is not possible since there is a path from C to E and E is a $Use(B)$. A detailed discussion of NSE checking is provided in Section 4.2.3.

—**Bidirectional NSE(u,v)** indicates that there is a NSE in both directions, (u, v) and (v, u). As shown in Figure 6-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 has a bidirectional NSE so that all variables in the clique must be alive simultaneously. Formally, this relationship can be stated as $u, v \in ES : \forall u, v, \exists\ NSE(u, v)$ & $NSE(v, u)$. There are three NSE cliques, $\{B, C, D\}$, $\{F, G\}$ and $\{I\}$, shown in the example in Figure 6-c. A single variable is a degenerate case of an NSE clique.

—**Partition(E0,E1)** represents a bi-partitioning of the excessive set ES such that $E0 \cap E1 = \emptyset$ and $E0 \cup E1 = ES$. In Figure 6-d, the excessive set is partitioned into $E0 = \{I\}$, $E1 = \{B, C, D, F, G\}$.

—**Pred_set(E1)** is the set of nodes of $Pred(u)$, where u is a variable in $E1$, that are not in the excessive set. Formally, $v \in Pred(u) : u \in E1$ & $v \notin ES$. In Figure 7-a, $Pred\_set(E1)$=$\{A\}$.

—**Succ_set(E0)** is the set of nodes of $Succ(u)$, where u is a variable in $E0$, that are not in the excessive set. Formally, $v \in Succ(u)$: $u \in E0$ & $v \notin ES$. In Figure 7-a, $Succ\_set(E0)$=$\{J\}$.

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 6-d 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 7-a is not a good candidate since $|Succ\_set(E0)| = |E0| = 1$.

4.2.2 *Pre-partitioning: NSE construction.* Prior to the two partitioning steps, non-serializable edges between variables must be identified. As shown in Figure 4, NSEs are created for variables which are code segment inputs (outputs), since they cannot be serialized after (before) other ES variables. Additionally, a breadth first search is performed to locate nodes which are successors of one variable in ES

Given
  $G(V, E)$: a DDG with directed edges.
  ES: an excessive set.

Produce
  E_{nse}: a set of non-serializable edges

```
Procedure NSEConstruct
    for each edge (u, v) ∈ E
        Succ(u) ← Succ(u) ∪ v
        Pred(v) ← Pred(v) ∪ u
    endfor
    for each node u ∈ ES
        if |Pred(u)| = 0 /* If node is graph input */
            E_nse ← E_nse ∪ NSE(v, u), ∀ v ∈ (ES − u)
        else
            From node u, apply breath first search in G
                Desce(u) ← all discovered nodes ∈ V
            for each node v ∈ (ES − u)
                if |Succ(v)| = 0 /* If node is graph output */
                    E_nse ← E_nse ∪ NSE(v, u)
                else if Desce(u) ∩ Succ(v) ≠ ∅
                    E_nse ← E_nse ∪ NSE(v, u)
                endif
            endfor
        endif
    endfor
end Procedure
```

Fig. 4. NSE construction

and descendents of another variable in ES. These ES variables also require NSEs because they cannot be serialized.

4.2.3 *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.

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 6-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 $\{I\}$). They are required by operations E, F, G and J respectively. Subsequently, the two largest non-overlapping NSE cliques, $\{B, C, D\}$ and $\{I\}$, are selected from the candidates. Detailed steps used for NSE clique generation are presented in Figure 5.

After NSE clique generation, the coarsening step merges two NSE cliques together

```
Given
    G(V, E): a DDG with directed edges.
    ES: an excessive set.

Produce
    Clique_sel: a minimum-size set of NSE cliques that covers ES

Procedure CliqueGenerate
    /* Intermediate values */
    Clique_cand: a set of candidate NSE cliques
    Q: Queue which holds u ∈ V during graph traversal
    done[]: array which indicates node u ∈ V has been visited

    /* Form clique from all graph outputs */
    Clear Clique_cand, Q, done[]
    for each node u ∈ ES,
        if |Succ(u)| = 0
            done[u] = YES; Q.push(u)
            clique ← clique ∪ u
        endif
    endfor
    Clique_cand ← Clique_cand ∪ clique

    /* Find cliques to cover ES */
    while Q ≠ ∅
        u ← Q.pop()
        for each node v ∈ Pred(u) /* Backward graph traversal */
            if v ∈ ES
                clique ← clique ∪ v
            endif if done[v] == NO
                done[v] = YES; Q.push(v)
            endif
        endfor
        Clique_cand ← Clique_cand ∪ clique
    endwhile

    Uncovered_nodes: nodes in ES not covered by a clique
    partition_ind[]: clique indices for each node u in ES

    /* Select minimum-sized set of cliques that covers ES */
    Uncovered_nodes ← ES, j = 0
    while Uncovered_nodes ≠ ∅
        loop over all cliques ∈ Clique_cand
            clique ← clique i ∈ Clique_cand which maximizes |Clique_cand[i] ∩ Uncovered_nodes|
        endloop
        for each node u ∈ clique
            partition_ind(u) = j
        endfor
        Uncovered_nodes = Uncovered_nodes - clique
        Clique_sel[j] = clique; j++
    endwhile
end Procedure
```

Fig. 5. NSE clique generation

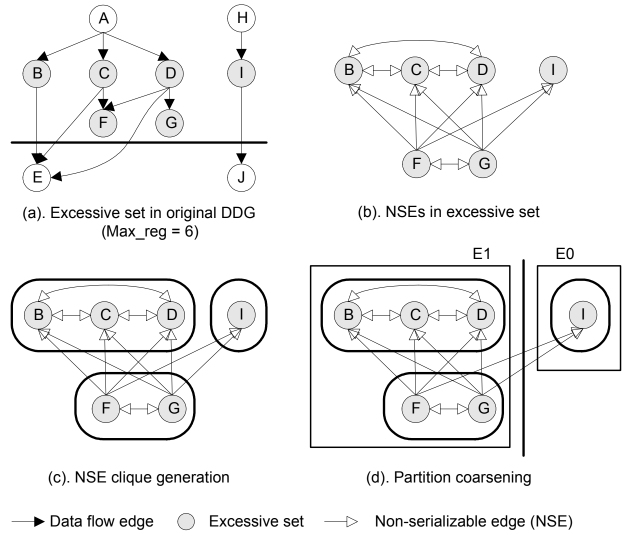

Fig. 6. Partitioning: coarsening step

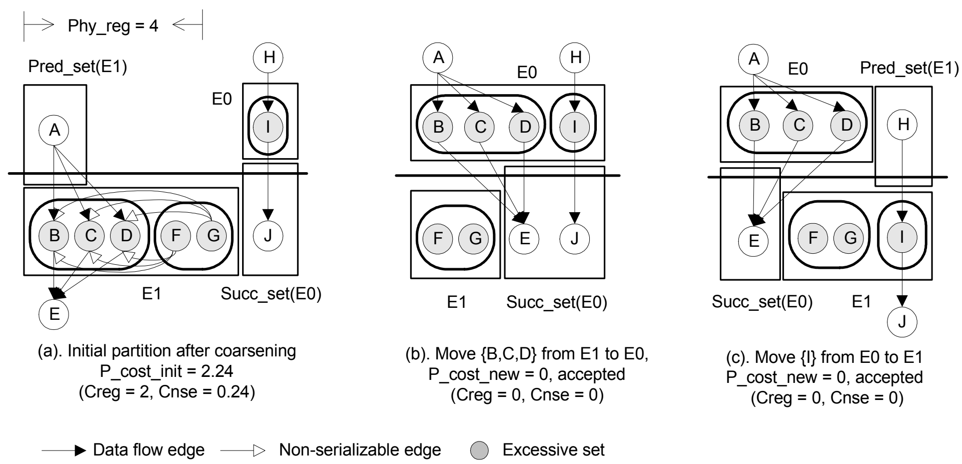

Fig. 7. Partitioning: refinement step

based on the number of NSE edges between them. As shown in Figure 6-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 6-c, after coarsening, the initial partition is $E0 = \{I\}$ and $E1 = \{B, C, D, F, G\}$. Detailed steps used for coarsening are presented in Figure 8.

4.2.4 *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:

$$P\_cost = C_{reg} + C_{NSE}; \tag{1}$$

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

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

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

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

As shown in Figure 7-a, the example partition has the following topology: $|E0| = 1$, $|Pred\_set(E1)| = 1$; $|E1| = 5$, $|Succ\_set(E0)| = 1$. Assuming $Phy\_reg = 4$, then C_{reg} = Max(2,6) - 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.

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

The second part, which is based on $NSE(E0)$ and $NSE(E1)$, is the number of directional NSE between NSE cliques in $E0$ and $E1$, respectively. This part is only effective when C_{reg} is positive, indicating another round of reduction is required to avoid spills. To allow for further reduction, $NSE(E0)$ and $NSE(E1)$ should also be minimized. To estimate the effect of $NSE(E0)$ and $NSE(E1)$, scalar factors $\beta_0 = (0.1 \times C_{reg})/|E0|$ and $\beta_1 = (0.1 \times C_{reg})/|E1|$ are applied. The denominators in α, β_0, and β_1 indicate a bias towards unbalanced partition sizes. Large $|E0|$ or $|E1|$ sets can be more easily reduced in later iterations of the Tetris algorithm. The 0.1 factor was determined via experimentation. The value indicates the relative importance of inter-partition versus intra-partition NSEs.

For the initial partition shown in Figure 7-a, C_{reg} = 6 - 4 = 2, $C_{NSE} = \beta_1 \times NSE(E1) = 0.24$ and the initial cost value P_cost_init = C_{reg} + C_{NSE} = 2.24. Note that NSE(E0) is 0 since there are no non-serializable edges in E0 and NSE(E0,E1) is 0 since there are no non-serializable edges between E0 and E1. To minimize the partition cost, a refinement step un-coarsens partitions and randomly moves NSE cliques from $E1$ to $E0$, then $E0$ to $E1$. In general, if the new partition cost, P_cost_new, is smaller than the initial partition cost, P_cost_init, then the move is accepted. The partition snapshot with the smallest P_cost is recorded and chosen as the final partition. Detailed steps used for refinement are presented in Figure 9.

Given
$G(V, E)$: a DDG with directed edges.
ES: an excessive set.
$Clique_{sel}$: a minimum-size set of NSE cliques that covers ES

Produce
$E0$, $E1$: Partitioning of ES
$partition\_ind$[]: Partition assignment indices for cliques
$NSE(E0, E1)$, $NSE(E0)$, $NSE(E1)$: Sets of non-serializable edges between/within partitions

Procedure PartitionCoarsen
/* Intermediate values */
NSE_count: two dimension array recording NSE count between/within partitions
$Partition$: Working set of clique groupings

/* Merge cliques until only two partitions are left */
$Partition = Clique_{sel}$
while |Partition| > 2
Initialize NSE_count[][] to 0
for each $NSE(u, v) \in E_{nse}$
$NSE\_count[partition\_ind[u], partition\_ind[v]]{+}{+}$
endfor
Select i, j that maximizes ($NSE\_count[i][j]$+$NSE\_count[j][i]$)
$Partition[i] \leftarrow Partition[i] \cup Partition[j]$
for each node u *in* $Partition[j]$
$partition\_ind[u] = i$
endfor
endwhile

$E0 \leftarrow Partition[0]$; $E1 \leftarrow Partition[1]$
$NSE(E0, E1) = NSE\_count[0][1]$
$NSE(E0) = NSE\_count[0][0]$; $NSE(E1) = NSE\_count[1][1]$
end Procedure

Fig. 8. Partitioning coarsening

As shown in Figure 7-a, partition $E1$ contains two NSE cliques, $\{B, C, D\}$ and $\{F, G\}$. Partition $E0$ contains only one NSE clique, $\{I\}$. The refinement of this example is described below:

(1) Move $\{B, C, D\}$ from $E1$ to $E0$ as shown in Figure 7-b. This move reduces C_reg to 0 and C_NSE to 0. Because P_cost_new (0) $<$ P_cost_init (2.24), this move is accepted.
(2) Move $\{I\}$ from $E0$ to $E1$ as shown in Figure 7-c. Because this move does not change C_{reg} and C_{NSE}, it is also accepted.

In this example, the partition in Figure 7-c has a minimum P_cost of 0. Therefore, the final partition is $E0 = \{B, C, D\}$ and $E1 = \{F, G, I\}$.

4.2.5 *Partitioning: complexity analysis.* The complexity of each stage of partitioning can be characterized based on V, the number of variables (nodes) in graph $G(V, E)$. NSE construction requires nested loops. The outer loop requires

```
Given
    G(V,E): a DDG with directed edges.
    E0, E1: Partioning of ES
    NSE(E0,E1), NSE(E0), NSE(E1): Non-serializable edges between/within partitions
    partition_ind[]: Partition assignment indices for nodes
Produce
    updated E0, E1: Partitioning of ES

Procedure PartitionRefinement

    /* Determine predecessor and successor sets for each clique */
    for each node u ∈ ES
        for each incoming edge (v,u) ∈ E
            Pred_set[partition_ind[u]] ← Pred_set[partition_ind[u]] ∪ v
        endfor
        for each outgoing edge (u,v) ∈ E
            Succ_set[partition_ind[u]] ← Succ_set[partition_ind[u]] ∪ v
        endfor
    endfor

    /* Intermediate values */
    Pred_set: Set of nodes Pred(u), where u is a node in a partition
    Succ_set: Set of nodes Succ(u), where u is a node in a partition
    P_cost_min: Minimum partition cost (P_cost)

    /* Consider successors or predecessors not in ES */
    Pred_set(E1) ← Pred_set[1] - (Pred_set[1] ∩ ES)
    Succ_set(E0) ← Succ_set[0] - (Succ_set[0] ∩ ES)
    Calculate Max_reg using Equation (3)
    Calculate P_cost(E0,E1) using Equation (1)
    P_cost_min = P_cost(E0,E1)

    /* Swap cliques to find lower cost partitioning */
    for each clique ∈ E0
        E0' ← E0 - clique
        E1' ← E1 + clique
        Based on (E0',E1'), update NSE_count, Pred_set[] and Succ_set[]
        Calculate P_cost(E0',E1') using Equations(1-4)
        if P_cost(E0',E1') < P_cost_min
            P_cost_min = P_cost(E0',E1')
            E0 ← E0'; E1 ← E1'
        endif
    endfor
    Repeat above for loop for clique movement E1 to E0
end Procedure
```

Fig. 9. Partitioning refinement

O($|ES|$) iterations since all nodes in the ES are considered. For each node, a breath first search, which takes O($|V|$ + $|E|$), is applied. The inner-loop, which requires O($|ES|$) iterations, checks the intersection of two sets, which takes O($|V|$). Therefore, the complexity of NSE construction is O($|ES|$ * ($|V|$ + $|E|$ + $|ES|$ * $|V|$)) = O($|V|^3$).

The NSE clique generation step first creates a clique of graph output nodes, O($|V|$). A backward traversal of the DDG is then performed, which requires O($|V|$ + $|E|$) steps. The final loop applies a greedy minimum-size set covering algorithm of O($|V|^3$) steps. Overall, the complexity of NSE clique generation is O($|V|^3$).

The coarsening step contains a loop with O($|ES|$) iterations. In each iteration, NSE_count calculation takes O($|E_{nse}|$). Maximum NSE_count selection therefore requires O($|ES|^2$). Since partition merging requires O($|ES|$), the complexity of the coarsening step is O($|ES|$ * ($|E_{nse}|$ + $|ES|^2$) = O($|V|^3$).

The refinement step first constructs $Pred\_set$ and $Succ\_set$, which requires O($|ES|$ + $|E|$) steps. During clique swapping, two loops are used. The first loop requires $|E0|$ iterations. During the loop, a clique is moved to $E1$ and a new cost is calculated. These actions require an update of NSE_count, $Pred\_set$ and $Succ\_set$ with a complexity of O($|E_{nse}|$ + $|ES|$ + $|E|$). Thus, the complexity of the first loop is O($|E0|$ * ($|E_{nse}|$ + $|ES|$ + $|E|$)). The second loop that moves cliques from $E1$ to $E0$ has similar complexity as O($|E1|$ * ($|E_{nse}|$ + $|ES|$ + $|E|$)). Overall, the complexity of the refinement step is O($|ES|$ * ($|E_{nse}|$ + $|ES|$ + $|E|$)) = O($|V|^2$).

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 $E1$.

As discussed in Section 3, **compatible serializations** represent a set of serializations that can be applied together without causing cycles which inhibit schedules. Since serialization ($D \rightarrow F$) (Figure 10-a) requires a serial edge (G to F) and serialization ($D \rightarrow G$) requires a serial edge (F to G), applying both serializations causes a cycle between F and G. Therefore, serializations ($D \rightarrow F$) and ($D \rightarrow G$) are not compatible and they cannot be applied simultaneously. Formally, serialization for ($u \rightarrow v$) cannot be applied if one of following conditions is true:

(1) $|Pred(v)| = 0$. This expression indicates that node v is an input node of the DDG. Because $Lv(v)$ starts from the beginning of the DDG, v cannot be serialized after other nodes. Therefore, $NSE(u, v)$ is inserted.

(2) $|Succ(u)| = 0$. This expression indicates that node u is an output node of the DDG. Because $Lv(u)$ does not end in the DDG, no other nodes can be serialized after u. Therefore, $NSE(u, v)$ must be inserted.

(3) $Succ(u) \cap Desce(v) \neq \emptyset$; $Desce(v)$ is a set of nodes: $w \in V$, where $\exists$ a path (v, w)

If the condition (3) is true, then ($u \rightarrow v$) will create a cycle. A proof of this condition appears in Appendix A in Lemma 1.

As discussed in Section 3, **compatible serializations** represent a set of serializations that can be applied together without causing cycles. Cycles caused by

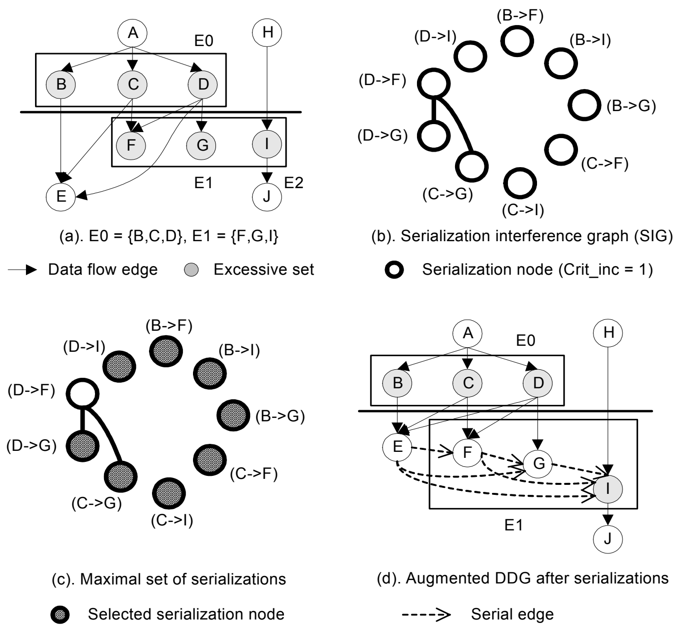

Fig. 10. Serialization step

incompatible serializations can inhibit any possible schedule. In order to select a maximal 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 $(E0, E1)$. $SIG(SV, IE)$ contains a set of serialization nodes (snodes) SV and a set of non-directed interference edges (iedges) IE. These terms are formally defined as:

—A **serialization node** represents a potential serialization $(u \rightarrow v)$ if there is no $NSE(u, v)$. As shown in Figure 10-a, the partition, $E0 = \{B, C, D\}$ and $E1 = \{F, G, I\}$, has no $NSE(E0, E1)$. Therefore, SV in Figure 10-b contains all 9 potential serialization nodes from $\{B, C, D\}$ to $\{F, G, I\}$. Formally, $SV \leftarrow (u \rightarrow v)$: $u \in E0$, $v \in E1$ & no $NSE(u, v)$).

—A **serialization interference edge** between two serialization nodes indicates that the nodes are incompatible and they cannot be applied together. Figure 10-b shows that there is an interference edge between snode $D \rightarrow F$ and snode $D \rightarrow G$ because they are not compatible.

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 serial edge ($Use(u)$ to v) and ($Use(s)$ to t), where $Use(u) \in Succ(u)$ and $Use(s) \in Succ(s)$. Such cycles can only exist if there is a path from t to $Use(u)$

```
/* Construct a SIG based on partition (E0,E1) */
Given
    G(V, E): a DDG with directed edges.
    E0, E1: Partioning of ES
    E_nse: set of non-serializable edges
Produce
    SV: set of serialization nodes (snodes) for graph G
    IE: set of interference edges (iedges) for graph G
    SIG(SV, IE): a graph with undirected edges

Procedure SIGConstruct
    SV = ∅; IE = ∅;
    for each node u ∈ E0
        E1' = E1; /* Make copy of E1 */
        /* Remove nodes that already have a non-serialization edge */
        for each edge NSE(u, v) ∈ E_nse
            E1' = E1' - v;
        endfor
        for each node v ∈ E1'
            SV ← SV∪ snode(u → v)
        endfor
    endfor

    for each pair (snode1, snode2) ∈ SV
        /* Compatibility check involves check for two NSEs and two node comparisons loops */
        if compatibility check in Section 4.3.1 for snode1, snode2 is not satisfied
            IE = IE + iedge(snode1, snode2)
        endif
    endfor
end Procedure
```

Fig. 11. SIG construction

and another path from v to $Use(s)$. A path from t to $Use(u)$ indicates either there is a $NSE(u,t)$ or t is a $Use(u)$. Similarly, a path from v to $Use(s)$ indicates either there is a $NSE(s,v)$ or 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 **compatibility check** conditions is true

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

A proof of these conditions appears in Appendix A in Lemma 2. Detailed steps used for SIG construction are presented in Figure 11.

4.3.2 *Maximal Serializations.* Since two compatible serialization nodes are not connected (independent) in a SIG, determining the maximal 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

[Cormen et al. 1990]. 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 $Crit\_inc$.

$$S\_cost = \gamma \times N\_deg + Crit\_inc; \tag{5}$$

The N_deg term is the SIG node degree, the number of serialization interference edges connected to the node. As shown in Figure 10-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 non-negative critical path increase caused by serial edges. A serial edge from a $Use(u)$ to v increases the critical path by:

$$Crit\_inc(Use(u)\ to\ v) = Max((Etime(Use(u)) - Ltime(v) + 1), 0); \tag{6}$$

where $Etime/Ltime$ represents the earliest/latest time a node can be scheduled without increasing the DDG critical path. The earliest time ($Etime$) of an operation is calculated by a forward graph traversal using the following equation:

$$Etime(v) = Max(Etime(w) + Delay(w)); \tag{7}$$

where w are variables required to calculate v. $Delay(w)$ represents the delay of operation w. For demonstration purposes, all operations in example DDGs have the same delay of once clock cycle. As shown in Figure 10-a, $Etime$ of E depends on $Etime$ and $Delay$ of three operations, B, C and D because $\{B, C, D\}$ are required to calculate E. Similarly, the latest time ($Ltime$) of an operation is calculated by a backward graph traversal using the following equation:

$$Ltime(v) = Min(Ltime(Use(v)) - Delay(v)); \tag{8}$$

$Etime$ and $Ltime$ values can be determined using a graph slack analysis algorithm [Marquardt et al. 2000], which requires a forward and backward graph traversal of the DDG. For a serialization node ($u \rightarrow v$) requiring multiple serial edges, the critical path increase is decided as:

$$Crit\_inc(u \rightarrow v) = Max(Crit\_inc(Use(u)\ to\ v)); \tag{9}$$

As shown in Figure 10-a, the critical path of the original DDG is A-B-E. Based on Equation 7 and 8, A and H have ($Etime$, $Ltime$) of (1, 1). B, C, D and I have ($Etime$, $Ltime$) of (2, 2). E, F, G and J have ($Etime$, $Ltime$) of (3, 3).

For a serialization node ($B \rightarrow F$) shown in Figure 10-b, only one serial edge (E to F) is required. Based on Equation 6, this serial edge increases the critical path by $Max(\ (Etime(E)\ \text{-}\ Ltime(F) + 1),\ 0\ ) = 1$. The DDG critical path is increased from A-B-E to A-B-E-F. Based on Equation 9, $Crit\_inc$ for serialization node $(B \rightarrow F)$ is equal to 1. Similarly, other serialization nodes in Figure 10-b can be calculated using above equations. In this example, 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 selection is first biased towards N_deg values of 0, followed by N_deg values of 1 with minimal $Crit\_inc$ values. 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 three times of the delay of a memory access operation, it is not applied, regardless of N_deg.

The SIG in Figure 10-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 v) are inserted into the DDG. A detailed outline of this step is presented as part of Figure 12.

For the example in Figure 10-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:

—$\{B \rightarrow F\}$ requires a serial edge (E to F).
—$\{B \rightarrow I\}$ requires a serial edge (E to I).
—$\{B \rightarrow G\}$ requires a serial edge (E to G).
—$\{C \rightarrow F\}$ requires a serial edge (E to F).
—$\{C \rightarrow I\}$ requires 2 serial edges, (E to I) and (F to I).
—$\{D \rightarrow I\}$ requires 3 serial edges, (E to I), (F to I) and (G to I).

2. Select 2 serialization nodes with N_deg of 1:

—$\{D \rightarrow G\}$ requires 2 serial edges (E to G) and (F to G).
—$\{C \rightarrow G\}$ requires 2 serial edges (E to G) and (F to 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 10-d, after applying the above 8 serializations, the augmented DDG contains 6 serial edges. Max_reg of the augmented DDG is reduced from 6 to 4 with a critical path increase of 4. The new excessive set is $\{B, C, D, I\}$ and the new critical path is A-B-E-F-G-I-J.

A limitation of the Tetris heuristic is that it only focuses on reducing the maximum register requirement. As a consequence, Tetris may cause a significant increase in the DDG critical path, which may limit the final performance. In the above example, a register reduction of 2 is achieved at the cost of a large critical path increase of 4. To address this limitation, we present an enhanced heuristic, Tetris-XL, in Section 5. Details of serialization selection are included in Figure 12. In general, Tetris does not guarantee a reduction in Max_reg. For example, a collection of node pairs in an initial DDG, with each pair connected by a single

```
Given
    G(V, E): a DDG with directed edges.
    ES: an excessive set.
    SV: serialization nodes (snodes)
    IE: interference edges (iedges)
    SIG(SV, IE): a graph with undirected edges

Produce
    snode_sel: subset of SIG(SV, IE) representing selected serializations
    SE: set of serial edges to be added to DDG G

Procedure SnodeSelect
    Determine etime(u) and ltime(u) values for each u ∈ ES
        using slack analysis algorithm [Marquardt et al. 2000]
    /* Calculate serialization cost */
    S_cost: cost of an snode in SV
    /* Evaluate performance loss from serial edges. */
    for each snode ∈ SV
        u = From(snode); v = To(snode)
        for each node w ∈ Succ(u)
            Calculate Crit_inc(w to v) using Equation 6
        endfor
        Crit_inc(snode) = MAX(Crit_inc(w to v))
        Calculate S_cost using Equation 5
    endfor

    /* Intermediate values */
    S_sort: List of serialization snodes sorted by S_cost
    Snode_conflict: serializations which cannot be used due to conflicts

    /* Select serializations and identify required serial edges. */
    S_sort ← Sort SV by S_cost
    while |S_sort| ≠ ∅
        snode = Head(S_sort)
        if snode ∉ Snode_conflict
            snode_sel ← snode_sel ∪ snode
            p ← From(snode); q ← To(snode)
            for each node r ∈ (Succ(p) - q)
                SE ← SE ∪ sedge(r to q)
            endfor
            for each iedge (snode, snode1) ∈ IE
                Snode_conflict ← Snode_conflict ∪ snode1
            endfor
        endif
    endwhile
end Procedure
```

Fig. 12. Serialization selection

directed edge, cannot be serialized if the edge sources are graph inputs and the sinks are graph outputs.

4.3.3 *Serialization: complexity analysis.* In the SIG construction step, serialization node generation requires O($|E_{nse}|$ + |E0|*|E1|) steps. The interference edge generation evaluates O($|E0|^2$*$|E1|^2$) snode pairs. For each snode pair, the compatibility check takes O(|1|). Therefore, the complexity of the SIG construction step is O($|E0|^2$*$|E1|^2$) = O($|V|^4$), although in most cases the size of the excessive set is much smaller than the total set of DDG nodes.

The *S_cost* calculation step first applies a topological ordering of G, which takes O(|V| + |E|) steps. The *etime* calculation applies a forward traversal of G, which takes O(|V| + |E|) steps. Similarly, the *ltime* calculation applies a backward traversal of G, which also takes O(|V| + |E|) steps. The *S_cost* calculation takes O(|SV| + |IE|) = O(|E0|*|E1| + $|E0|^2$*$|E1|^2$) steps. Therefore, the complexity of the *S_cost* calculation step is O($|E0|^2$*$|E1|^2$) = O($|V|^4$) .

The serialization selection step first applies a sort, which requires O($|SV| ln |SV|$) steps. Then, the selection step takes O(|SV| + |IE|) = O(|E0|*|E1| + $|E0|^2$*$|E1|^2$) steps. Therefore, the complexity of the serialization selection step is O($|E0|^2$*$|E1|^2$) = O($|V|^4$).

4.3.4 *Serializations: Comparison to URSA.* Although there are significant differences, URSA provides a serialization approach which is somewhat similar to Tetris. Like Tetris, URSA first determines the excessive set of a code block. To reduce the excessive set, URSA moves an operation to a separate *resource hole*. If an operation is the last operation in the live range of several variables, this action may free up several registers. A limitation of the approach is that it only considers moving an individual operation at each step. It is observed that in many situations, it could be more beneficial to apply serializations in groups rather than as individual operations [Berson et al. 1998]. Compared with URSA, Tetris tries to apply a group of serializations by moving a partition of nodes in the excessive set to a position after other nodes. This approach may be especially beneficial when two parts of the excessive set are relatively independent.

5. DELAY REDUCTION VIA TETRIS-XL

The main difference between Tetris and Tetris-XL is shown in Figure 13. As discussed in Section 4, the Tetris partitioning heuristic includes the following steps. The first partitioning step is coarsening, which generates the initial partitions ($E0$, $E1$) while minimizing the number of non-serializable edges (NSEs) from $E0$ to $E1$. The second partitioning step is refinement, which moves variables between partitions $E0$ and $E1$ to reduce the maximum register requirement. Because Tetris refinement only considers register reduction, subsequent serialization may significantly increase the DDG critical path.

To address this limitation, Tetris-XL applies an alternate refinement step as shown in Figure 13. The Tetris-XL refinement evaluates both the maximum register requirement *and* the critical path increase. The coarsening and serialization steps are the same for both Tetris and Tetris-XL.

To demonstrate the basic idea of Tetris-XL, a simple example is presented in

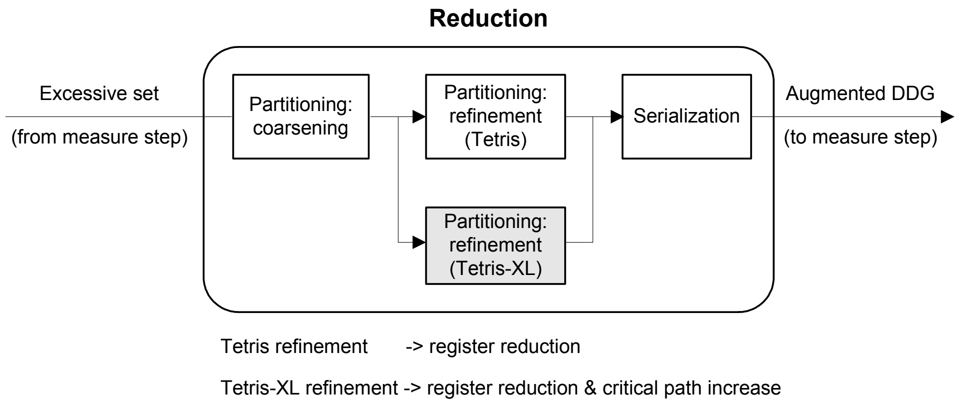

Fig. 13. Tetris and Tetris-XL flow

Figure 14. The example DDG in Figure 14-a contains 7 operations. Assume all operations have unit delay so that the original critical path is 3 (e.g. A-B-D). As shown in Figure 14-b, when operations D and I are scheduled last, variables $\{B, C, H, F\}$ are alive simultaneously and the maximum register requirement is 4. If only 3 physical registers are available, variables $\{B, C, H, F\}$ form an excessive set, which requires a reduction step.

As shown in Figure 14-c, Tetris bi-partitions the excessive set into $E0 = \{B, C\}$ and $E1 = \{F, H\}$. After serializing $\{F, H\}$ after $\{B, C\}$, the maximum register requirement is reduced from 4 to 3. For the example shown in Figure 14-c, the augmented DDG has a maximum register requirement of 3 $\{D, F, H\}$ and a critical path of 6 (A-B-D-F-H-I). Since Tetris-XL considers both register reduction and critical path control, Tetris-XL removes variables from serializations if they increase the critical path or do not contribute to register reduction. These removed variables form a separate partition $E2$, which stays unchanged after serialization. For example, Figure 14-d shows that variable H is removed from serialization, the critical path is reduced and register reduction is maintained. If $\{F\}$ is serialized after $\{B, C\}$, the augmented DDG requires the same maximum register count of 3 $\{D, F, H\}$ and exhibits a smaller critical path of 4 (A-B-D-F), leading to better performance.

In the next section, the partition cost function used in the Tetris-XL refinement is discussed. The cost function is then applied to the same refinement example used to describe Tetris.

5.1 Tetris-XL Partition Cost

In Tetris refinement, variables are moved between partitions $E0$ and $E1$ to reduce the partition cost, P_cost, which represents the maximum register requirement. Tetris-XL moves variables between $E0$ and $E1$ based on a new cost function, P_cost_XL, which takes both critical path increase and the maximum register requirement into account. If a variable in $E0$ or $E1$ does not contribute to register reduction, Tetris-XL moves it to a separate partition, $E2$. The same action is performed if a variable in $E0$ or $E1$ causes a large critical path increase. The Tetris-XL

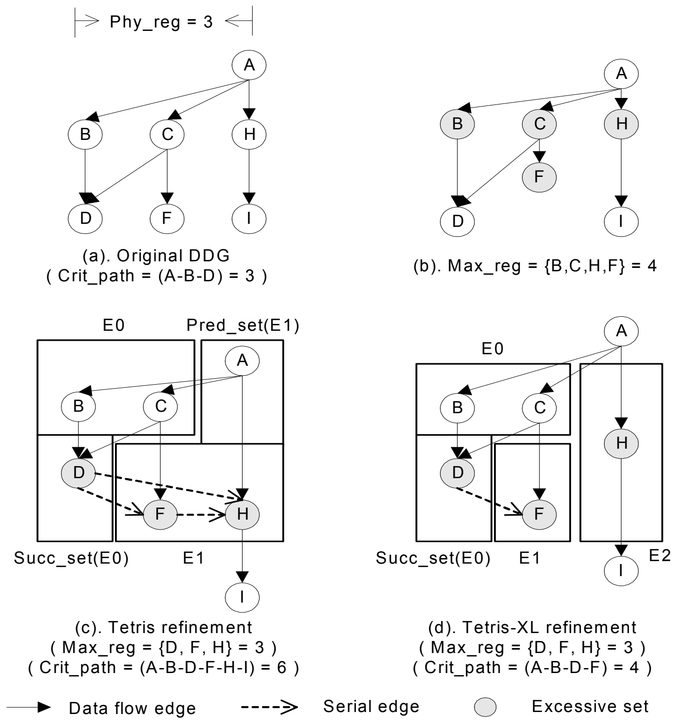

Fig. 14. Tetris refinement versus Tetris-XL refinement

cost function, P_cost_XL, is defined as:

$$P\_cost\_XL = C_{reg\_XL} + C_{NSE} + C_{crit\_XL}; \tag{10}$$

Compared with the Tetris cost function, P_cost, shown in Equation 1, P_cost_XL adds a new C_{crit_XL} term that evaluates the effects of critical path increase. To consider the effect of the new partition $E2$ on the maximum register requirement, Tetris-XL also applies a C_{reg_XL} term to replace the C_{reg} term in the previous Tetris cost function. The original C_{NSE} term used in Tetris is kept unchanged in Tetris-XL.

Figure 15 shows the calculation of the P_cost_XL for two partitions with $C_{NSE} = 0$. Both partitions are taken from the example in Figure 14. The C_{reg_XL} term represents the non-negative gap between the maximum register requirement, Max_reg_XL and available physical registers, Phy_reg.

$$C_{reg\_XL} = Max((Max\_reg\_XL - Phy\_reg), 0); \tag{11}$$

In Tetris-XL, variables in partition $E2$ can be alive simultaneously with either $E0$ or $E1$. Therefore, the Max_reg_XL term is calculated based on the topology of $E0$, $E1$ and $E2$.

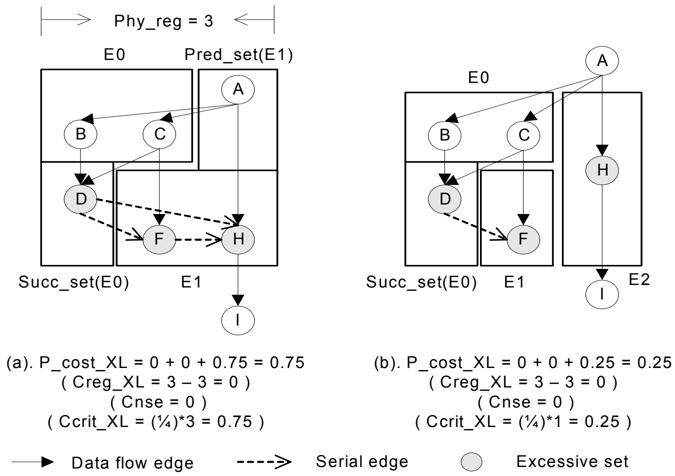

Fig. 15. Tetris-XL partition cost

$$Max\_reg\_XL = Max((|E0| + |Pred\_set(E1)|), (|E1| + |Succ\_set(E0)|)) + |E2|; \tag{12}$$

As shown in Figure 15-b, the example partition has the following topology, $|E0|$ = 2, $|Succ\_set(E0)|$ = 1, $|E1|$ = 1, $|Pred\_set(E1)|$ = 0 and $|E2|$ = 1. Based on Equation 12, Max_reg_XL = Max(2, 2) + 1 = 3. Similarly, the partition in Figure 15-a has Max_reg_XL = Max(3, 3) + 0 = 3. Because $Phy\_reg = 3$, both partitions have the same C_{reg_XL} = 3 - 3 = 0.

The second new term, C_{crit_XL} estimates the DDG critical path increase caused by all serializations and evaluates its adverse effect on final performance.

$$C_{crit\_XL} = \rho \times Tot\_crit\_inc; \tag{13}$$

In Equation 13, Tot_crit_inc represents the critical path increase in terms of clock cycles. As shown in Figure 15-b, the partition requires a serial edge (D to F), which increases the critical path from A-B-D to A-B-D-F. Based on the assumption that all operations in the DDG have one clock cycle delay, the critical path is increased by one clock cycle. If a DDG contains operations with various delays, Equation 6 can be applied to calculate the critical path increase. When multiple serial edges are required, the critical path increase of consecutive serial edges is accumulated. Tot_crit_inc is decided by the maximum accumulated critical path increase. As shown in Figure 15-a, the partition requires three serial edges. Serial edge (D to F) increases the critical path by one clock cycle (from A-B-D to A-B-D-F). Serial edge (D to H) increases the critical path by two clock cycles (from A-B-D to A-B-D-H-I). Serial edge (F to H) also increases the critical path by two clock cycles (from A-B-D to A-C-F-H-I). Finally, two consecutive serial edges, (D to F) and (F to H) determine that Tot_crit_inc = 1 + 2 = 3 clock cycles (from A-B-D to

A-B-D-F-H-I).

To decide whether a serialization can improve performance, Tetris-XL considers both the benefit of register reduction and the cost of critical path increase. For example, the serialization in Figure 15-a reduces the maximum register requirement by 1 but increases the critical path by 3 clock cycles. The register reduction of 1 saves one spill. As a consequence, it eliminates at least one memory store and one memory load operation. Assuming that both store and load operations have the same delay, $Mem\_delay = 2$ clock cycles, the register reduction of 1 may reduce the critical path by $2 \times Mem\_delay = 4$ clock cycles. Because the benefit of register reduction (4 clock cycles) overweighs the cost of critical path increase (3 clock cycles), the serialization is likely to improve the performance.

To make a tradeoff between register reduction and critical path increase, Equation 13 applies a constant scalar factor $\rho = 1/(2 \times Mem\_delay)$. For the partition in Figure 15-a, it has been shown that $Tot\_crit\_inc = 3$ so that $C_{crit\_XL} = 1/(2 \times 2) \times Tot\_crit\_inc = 1/4 \times 3 = 0.75$. Based on Equation 10, the partition in Figure 15-a has $P\_cost\_XL = C_{reg\_XL} + C_{NSE} + C_{crit\_XL} = 0 + 0 + 0.75 = 0.75$. For the partition in Figure 15-b with Tot_crit_inc of 1, $C_{crit\_XL} = 1/4 \times 1 = 0.25$ and $P\_cost\_XL = 0 + 0 + 0.25 = 0.25$. Therefore, Tetris-XL selects the partition in Figure 15-b with the smaller P_cost_XL of 0.25.

5.2 Tetris-XL Refinement

Tetris-XL uses the cost metrics shown in Equations 10 through 13 and the algorithm shown in Figure 9 to reduce the partition cost, P_cost_XL. In general, Tetris-XL un-coarsens the initial partitions and randomly moves NSE cliques between $E1$, $E0$ and $E2$. If the partition cost after a move, $P\_cost\_XL\_new$ is smaller than the initial partition cost, $P\_cost\_XL\_init$, then the move is accepted. The partition snapshot with the smallest P_cost_XL is recorded and chosen as the final partition.

Figure 16 shows the Tetris-XL refinement applied to the example illustrated in Figure 7. Because Tetris-XL applies the same coarsening step as Tetris, the same initial partition is generated, as shown in Figure 16-a. In the initial partition, the maximum register requirement is 6 so that C_{reg_XL} = 6 - 4 = 2. Tetris-XL uses the C_{NSE} term defined in Equation 4 so the value remains 0.24. Serializing $\{B, C, D, F, G\}$ after $\{I\}$ requires 5 serial edges from J, which causes a Tot_crit_inc of 2 (from A-B-E to H-I-J-B-E). Assuming the memory access delay is 2 clock cycles, $C_{crit\_XL} = 1/(2 \times 2) \times 2 = 0.5$. Therefore, the initial partition cost $P\_cost\_XL\_init$ = 2 + 0.24 + 0.5 = 2.74. The Tetris-XL refinement of this initial partition is described below.

(1) Move $\{B, C, D\}$ from $E1$ to $E0$ as shown in Figure 16-b. After this move, the maximum register requirement is reduced to 4 so that C_{reg_XL} = 4 - 4 = 0. C_{NSE} is reduced to 0 because there is no NSE from $E0$ to $E1$. Serializing $\{B, C, D, I\}$ after $\{F, G\}$ increases the critical path by 2 (from A-B-E to A-B-E-F-G) and $C_{crit\_XL} = 1/(2 \times 2) \times 2 = 0.5$. Because $P\_cost\_XL\_new$ (0.5) $\leq$ $P\_cost\_XL\_init$ (2.74), this move is accepted.

(2) Move $\{I\}$ from $E0$ to $E1$ as shown in Figure 16-c. After this move, C_{reg_XL} remains as 0 but the critical path is increased by 4 (from A-B-E to A-B-E-F-G-I-J) and $C_{crit\_XL} = 1/(2 \times 2) \times 4 = 1$. Therefore, $P\_cost\_XL\_new$ is increased

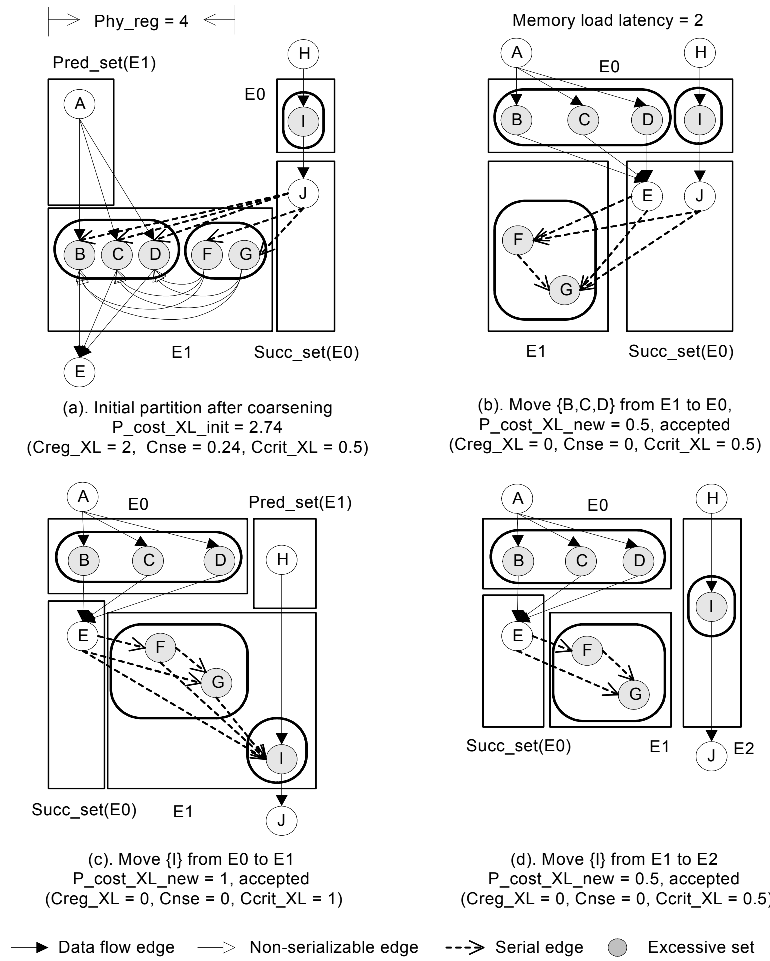

Fig. 16. Partitioning: refinement step (Tetris-XL)

to 1. Because $P\_cost\_XL\_new$ (1) is still smaller than $P\_cost\_XL\_init$ (2.74), this move is also accepted.

(3) Move $\{I\}$ from $E1$ to $E2$ as shown in Figure 16-d. After this move, C_{reg_XL} stays unchanged while the critical path increase drops to 2 (from A-B-E to A-B-E-F-G). $P\_cost\_XL\_new$ is reduced from 1 to 0.5 and the move is accepted.

In the above example , the partition in Figure 16-d is recorded as the final best partition with a smallest P_cost_XL of 0.5. The final partition is $E0 = \{B, C, D\}$, $E1 = \{F, G\}$ and $E2 = \{I\}$.

5.3 Tetris-XL Serialization

Tetris-XL applies the same serialization step as Tetris. First, a serialization interference graph (SIG) is generated as shown in Figure 17-b. The SIG includes 6 serialization nodes with the same $Crit\_inc$ of 1. The serialization node with the

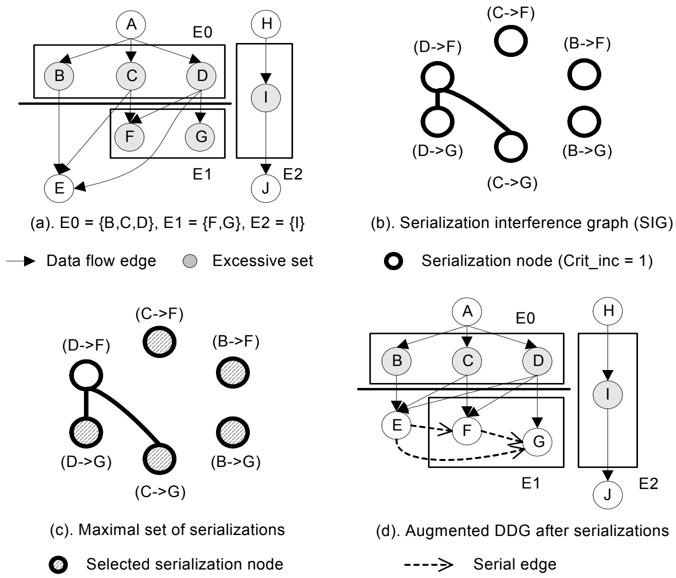

Fig. 17. Serialization step (Tetris-XL)

minimum number of interference edges, N_deg, is selected and applied first. The serialization process proceeds as follows:

1. Select 3 serialization nodes with N_deg of 0:

—$\{B \rightarrow F\}$ requires a serial edge (E to F).

—$\{B \rightarrow G\}$ requires a serial edge (E to G).

—$\{C \rightarrow F\}$ requires a serial edge (E to F).

2. Select 2 serialization nodes with N_deg of 1:

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

—$\{C \rightarrow G\}$ requires 2 serial edges (E to G) and (F to 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 17-d, after applying the above 5 compatible serializations, the augmented DDG contains 3 serial edges. The maximum register requirement of the augmented DDG is reduced from 6 to 4 with a critical path increase of 2. The new excessive set is $\{B, C, D, I\}$ and the new critical path is A-B-E-F-G. Compared with the Tetris result shown in Figure 10-d, Tetris-XL achieves the same register reduction of 2 with a smaller critical path increase of 2, resulting in better performance.

6. EXPERIMENTAL APPROACH AND RESULTS

To evaluate the effectiveness of Tetris and Tetris-XL reduction algorithms, a direct comparison to previous spill reduction techniques was performed. These techniques

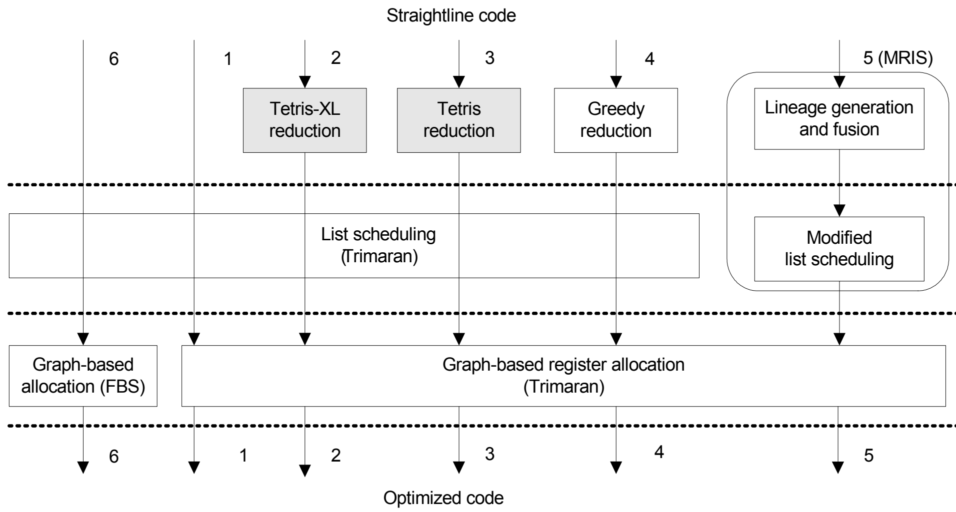

Fig. 18. Experimental flow in Trimaran framework

were implemented in an academic VLIW compiler, Trimaran, version 2.0 [Chakrapani et al. 2004]. 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 a set of programs taken from the Trimaran framework [Chakrapani et al. 2004] and three applications taken from the MediaBench suite [Lee et al. 1997]. Benchmarks *unepic*, *g721dec* and *mpeg2dec* are applications for image, audio and video signal processing, respectively.

As shown in Figure 18, our experiments include 6 flows. Flow 1 is the baseline Trimaran flow, which includes list scheduling followed by register allocation using graph-coloring. To control register pressure, flows 2, 3 and 4 apply Tetris, Tetris-XL and greedy serialization, respectively, before scheduling. Greedy serialization [Touati 2005] was discussed at the end of Section 3. Flow 5 applies another spill reduction technique, MRIS [Govindarajan et al. 2003], before the default graph-coloring register allocator in Trimaran. MRIS includes lineage generation/fusion and list scheduling as discussed in Section 2. MRIS was previously used [Govindarajan et al. 2003] to perform register allocation before instruction scheduling on out-of-order issue superscalar processors. We consider MRIS to be a good candidate for comparison to Tetris for VLIW processors since it has previously demonstrated a strong ability to perform schedule-sensitive register reductions. To limit the impact of false dependencies introduced by MRIS on performance, register reduction is only performed if the register requirement exceeds the number of available registers. Tetris and Tetris-XL also follow this restriction. In flow 6, an enhanced register allocator with FBS is applied after the default list scheduling in Trimaran. FBS is a frequency-based live range splitting technique [Kim 2001] described in Section 2.

All flows include dead code elimination, constant propagation, and loop unrolling prior to the steps shown in Figure 18. The results reported by Trimaran do not consider dynamic effects such as interrupts or cache interactions. Except where noted, the commercial architectures modelled by our experiments do not include caches, limiting the impact of this issue on the target architecture. Benchmark cycle counts are determined by Trimaran following compilation by assessing the schedule length of each code block, including cycles to handle spills, multiplied by the number of times each code block is called. The reported execution cycle counts are static cycle counts. In the experiments, multiply and memory load operations require two clock cycles. Other functional unit operations require one clock cycle.

6.1 Experiments with high register pressure

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 [Freescale Semiconductor, Inc. 2005]. These processors are often used in resource-constrained embedded systems. Our experiment evaluates the benefit of each individual technique for the 4-way architecture.

Table I compares **spills**, the total number of spill operations executed in each benchmark, as reported by Trimaran. A spill operation is a memory store or load operation inserted by the register allocation. Spills are shown in thousands of values. For the baseline flow, **spill ratio** is also presented as the percentage of **spills** to **ops**, the total number of operations (including spill operations) executed in each benchmark. A high spill ratio indicates that a benchmark suffers from high register pressure. At the bottom of the table, the geometric average (GEOMEAN) of spills is provided along with the geometric average of the per-benchmark percent changes versus the baseline Trimaran flow. Geometric average is used for averaging the spill counts due to the presence of widely varying absolute spill values.

Table II compares **cycles**, the total number of clock cycles required to execute each benchmark. Multiple operations are executed on each clock cycle. Cycles are also shown in thousands of values. At the bottom of the table, the geometric average of cycles is provided along with the geometric average of the per-benchmark percent changes versus the baseline Trimaran flow. Geometric average is used for averaging the cycle counts due to the presence of widely varying absolute cycle values.

For the baseline (flow 1) in Table I, on average, spills take up 46% of total executed operations, which indicates most benchmarks experience high register pressure. Among the five spill reduction techniques, Tetris-XL (flow 2), Tetris (flow 3) and MRIS (flow 4) are the three most efficient techniques in terms of reducing spills and improving performance. Compared with the baseline, Tetris-XL (flow 2), Tetris (flow 3) and MRIS (flow 4) reduce spills by 30%, 29% and 27%, respectively. As for the performance improvement, Table II shows that, on average, Tetris-XL, Tetris and MRIS reduce execution cycles by 30%, 25% and 16%, respectively. The execution cycle reduction of a VLIW program is decided by two factors, spill reduction and critical path increase. A large spill reduction with a small critical path increase leads to a large reduction in execution cycles.

It is observed that Tetris-XL, Tetris and MRIS reduce the maximum register requirement by reordering operations before instruction scheduling. Two additional

	Tetris-XL (flow 2)	Tetris (flow 3)	Greedy (flow 4)	MRIS (flow 5)	FBS (flow 6)	Baseline (flow 1)	
Benchmark	Spills (K)	Spills (K)	Spills (K)	Spills (K)	Spills (K)	Spills (K)	Spill ratio
bmm	132.2	195.7	195.7	190.3	189.4	195.7	59%
mm	146.2	143.1	178.0	175.0	182.6	175.0	56%
mm_double	128.1	168.1	168.1	168.4	171.5	168.1	55%
mm_dyn	161.8	123.9	238.3	207.4	252.5	252.9	66%
parms_test	4.8	6.9	6.8	4.2	7.8	6.9	44%
sqrt	3.0	2.2	4.1	3.9	4.4	4.0	54%
strcpy	6.9	7.8	8.5	3.5	8.6	16.0	45%
switch_test	2.5	2.6	2.6	2.6	2.6	2.6	14%
wave	20.6	19.3	29.8	13.5	29.8	35.5	58%
g721dec	111741.0	116112.0	143463.0	124823.0	135469.0	169149.0	30%
unepic	9253.0	7715.0	8892.0	10009.0	11153.0	11013.0	47%
mpeg2dec	201470.0	160276.0	244131.0	254589.0	244822.0	279593.0	58%
GEOMEAN	167.6	167.8	211.4	173.3	219.6	236.4	46%
% change	-30%	-29%	-11%	-27%	-7%		

Table I. Spills comparison on a VLIW with 16 registers and 4 FUs

	Tetris-XL (flow 2)	Tetris (flow 3)	Greedy (flow 4)	MRIS (flow 5)	FBS (flow 6)	Baseline (flow 1)
Benchmark	Cycles (K)	Cycles (K)	Cycles (K)	Cycles (K)	Cycles (K)	Cycles (K)
bmm	190.8	325.6	325.6	312.7	314.4	325.6
mm	200.5	215.9	292.3	295.7	330.6	300.6
mm_double	195.3	290.7	290.7	287.6	319.6	290.7
mm_dyn	237.7	190.5	332.8	287.1	358.3	348.7
parms_test	9.2	12.4	12.4	9.6	13.3	12.4
sqrt	5.4	4.7	7.4	7.7	8.8	8.0
strcpy	24.3	25.5	26.5	21.0	27.1	33.1
switch_test	13.0	13.0	13.0	13.5	13.0	13.0
wave	34.1	34.5	50.3	22.3	44.8	60.4
g721dec	236754.0	247411.0	270268.0	309737.0	227010.0	300607.0
unepic	15741.0	13715.0	14992.0	16639.0	18535.0	17944.0
mpeg2dec	159732.0	137828.0	216025.0	263829.0	313154.0	286892.0
GEOMEAN	302.1	321.3	391.0	360.8	418.6	429.4
% change	-30%	-25%	-9%	-16%	-3%	

Table II. Cycles comparison on a VLIW with 16 registers and 4 FUs

experiments illustrate the benefit of Tetris-XL over other approaches. Because spills are closely related to the maximum register requirement, a first experiment evaluates the maximum register requirement of each benchmark after spill reduction techniques are applied. For example, the first bar in Figure 19-a shows that Tetris-XL reduces the maximum register requirement of benchmark *bmm* by 44% versus the Trimaran baseline flow. Tetris-XL is able to reduce spills of benchmark *bmm* by 33%, as shown by the first bar of Figure 19-b. The geometric average of results in Figure 19-a shows that, on average, Tetris-XL, Tetris and MRIS reduces

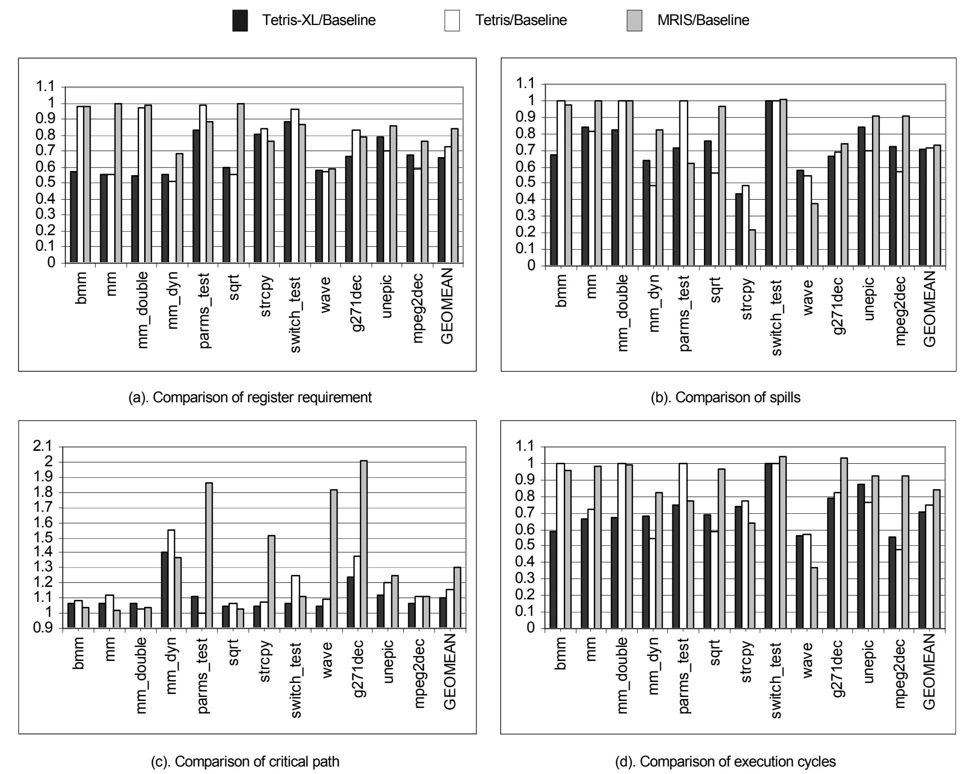

Fig. 19. Additional comparisons on a VLIW with 16 registers and 4 FUs

the maximum register requirement by 34%, 27% and 18%, respectively. These maximum register requirement reductions cause corresponding average spill reductions of 30%, 29% and 27% as shown in Figure 19-b.

The critical path increase caused by serialization plays an important role in deciding the final performance in terms of execution cycles. A second experiment compares the critical path of each benchmark after applying Tetris-XL, Tetris and MRIS. As shown in Figure 19-c, Tetris-XL, Tetris and MRIS increases the average critical path of benchmarks by 10%, 17% and 29%, respectively, compared with the baseline flow. Overall, Tetris-XL achieved a 30% reduction in execution cycles, outperforming both Tetris (25%) and MRIS (16%). Therefore, the benefit of Tetris-XL is a result of its ability to limit critical path increases during register reduction. In contrast, Tetris and MRIS do not consider the potential adverse effects of critical path increases caused by serial edges. The results in Figure 19 indicate that Tetris and Tetris-XL are effective in improving performance for a range of applications, including multimedia benchmarks. Applications with large data sets, such as scientific applications, which typically use a sizable number of registers, are likely to benefit more from our approach.

As currently structured, Tetris and Tetris-XL are optimized for application runtime reduction rather than reduced compile time. As noted in Section 4, both

Benchmark	no. lines	Tetris-XL (flow 2) (s)	Tetris (flow 3) (s)	Greedy (flow 4) (s)	MRIS (flow 5) (s)	FBS (flow 6) (s)	Baseline (flow 1) (s)
bmm	106	20.07	15.08	11.37	13.19	10.91	10.85
mm	48	16.36	10.52	9.51	11.30	8.23	8.18
mm_double	70	16.43	10.76	9.87	11.65	8.38	8.36
mm_dyn	49	19.85	11.39	10.57	13.02	9.95	9.87
parms_test	850	35.52	27.31	25.22	27.05	19.52	19.50
sqrt	34	26.31	6.53	5.85	6.30	5.48	5.45
strcpy	29	8.68	7.40	5.79	5.56	5.16	5.14
switch_test	357	9.54	9.27	9.10	10.50	8.44	8.41
wave	44	21.79	8.62	8.15	10.07	8.07	8.05
g721dec	1,490	231.10	221.77	94.61	109.53	46.27	45.04
unepic	2,697	2314.30	2152.12	582.84	612.87	449.98	419.74
mpeg2dec	8,680	1789.32	1540.07	288.61	377.28	142.38	141.38
GEOMEAN	211.50	48.58	32.72	21.78	24.92	17.52	17.31

Table III. Compile time statistics for VLIW with 16 registers and 4 FUs

	Tetris-XL (flow 2)	Tetris (flow 3)	Greedy (flow 4)	MRIS (flow 5)	FBS (flow 6)	Baseline (flow 1)
Spills	(K)	(K)	(K)	(K)	(K)	(K)
g721dec	81,609	78,618	97,375	91,256	102,750	108,280
unepic	2,580	2,406	4,154	4,051	4,731	5,843
mpeg2dec	21,620	20,182	40,693	38,157	40,283	81,267
GEOMEAN	120.46	124.01	165.48	133.83	172.01	195.15
% change	-38%	-36%	-15%	-31%	-12%	
Cycles	(K)	(K)	(K)	(K)	(K)	(K)
g721dec	287,605	293,071	301,412	297,054	307,191	320,055
unepic	9,694	10,019	10,121	10,188	11,171	12,319
mpeg2dec	158,029	163,347	168,210	161,894	170,450	175,895
GEOMEAN	294.10	322.05	374.10	331.29	391.24	401.86
% change	-27%	-20%	-7%	-18%	-3%	
Compile time	(s)	(s)	(s)	(s)	(s)	(s)
g721dec	81.54	79.73	65.57	71.34	29.31	26.48
unepic	128.80	125.90	109.75	120.75	71.00	66.50
mpeg2dec	414.09	397.71	260.93	298.03	113.90	105.21
GEOMEAN	30.99	21.19	18.22	20.31	14.19	13.86

Table IV. Statistics for VLIW with 16 registers and 4 FUs for restructured benchmarks

Tetris and Tetris-XL examine all possible serializations of variables in the excessive set. Multiple iterations of the algorithm are performed until no further Max_reg minimization can be achieved. This evaluation can lead to a compile time increase for designs with large code blocks. As shown in Table III, the large code blocks in the MediaBench benchmarks *g721dec*, *unepic*, and *mpeg2dec* lead to significant compile time increases versus baseline Trimaran compilation. The average code block size (about 75 lines) for these three benchmarks is two to three times the size

	Tetris-XL (flow 2)	Tetris (flow 3)	Greedy (flow 4)	MRIS (flow 5)	FBS (flow 6)	Baseline (flow 1)	
Benchmark	Spills (K)	Spills (K)	Spills (K)	Spills (K)	Spills (K)	Spills (K)	Spill ratio
bmm	20.0	28.8	28.8	28.3	27.8	28.8	18%
mm	21.3	20.6	44.0	40.6	40.8	40.9	23%
mm_double	19.3	26.3	26.3	26.2	26.3	26.3	16%
mm_dyn	39.7	30.3	77.1	65.3	57.7	81.1	38%
parms_test	2.1	2.3	2.3	2.1	2.3	2.3	21%
sqrt	0.3	0.2	0.7	1.0	0.7	1.0	23%
strcpy	0.0	0.0	0.1	0.0	0.1	0.1	1%
switch_test	2.5	2.6	2.6	2.6	2.6	2.6	14%
wave	2.3	1.4	11.1	0.4	12.3	21.4	46%
g721dec	35278.0	32610.0	39313.0	44484.0	43018.0	47487.0	26%
unepic	2592.0	2016.0	3362.0	3975.0	370.0	4168.0	11%
mpeg2dec	30563.0	5367.0	34726.0	70785.0	27178.0	72519.0	27%
GEOMEAN	27.5	21.9	41.9	34.0	41.1	50.9	15%
% change	-45%	-56%	-18%	-34%	-19%		

Table V. Spills comparison on a VLIW with 32 registers and 8 FUs

of the other benchmarks. For the three multimedia benchmarks, nearly half of this compile time increase is due to the measure step [Touati 2005; Berson et al. 1993] used to determine Max_reg after each Tetris iteration, not the Tetris algorithm itself. This compile time issue may be alleviated somewhat by limiting the use of Tetris and Tetris-XL to final pass compilation after code testing and optimization has been completed. In an additional experiment, the three large designs were restructured to achieve the same functionality, but with code block sizes reduced by about a factor of three. The results in Table IV demonstrate that the compile time can be reduced with continued runtime performance improvement versus the baseline flow. For the *unepic* and *mpeg2dec* designs, the baseline cycle count after restructuring improved on the initial coding. Baseline cycle count is slightly worse after recoding for *g721dec*. Geomean values in Table IV reflect averages across all designs, including those which have not been restructured.

6.2 Experiments with low register pressure

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 [Texas Instruments, Inc. 2000].

For a register size of 32, the baseline (flow 1) in Table V has a spill ratio of 15%. This value indicates a relatively low but non-trivial register pressure. Among the five spill reduction techniques, Tetris-XL (flow 2), Tetris (flow 3) and MRIS (flow 4) are still the most efficient techniques in terms of spill reduction. Compared with the baseline, Tetris-XL (flow 2), Tetris (flow 3) and MRIS (flow 4) reduce spills by 45%, 56% and 34%, respectively. As for performance improvement, Table VI shows that, on average, Tetris-XL, Tetris and MRIS reduce execution cycles by 19%, 15%

	Tetris-XL (flow 2)	Tetris (flow 3)	Greedy (flow 4)	MRIS (flow 5)	FBS (flow 6)	Baseline (flow 1)
Benchmark	Cycles (K)	Cycles (K)	Cycles (K)	Cycles (K)	Cycles (K)	Cycles (K)
bmm	99.8	104.5	104.5	107.4	101.8	104.5
mm	108.0	108.2	114.9	115.8	114.5	114.6
mm_double	100.8	101.5	101.5	104.8	101.4	101.5
mm_dyn	75.8	103.0	197.3	96.3	98.8	125.9
parms_test	6.1	6.1	6.1	7.1	6.1	6.1
sqrt	3.8	4.0	4.2	4.0	4.2	4.2
strcpy	18.2	18.8	18.3	20.6	18.3	18.3
switch_test	13.0	13.0	13.0	13.0	13.0	13.0
wave	12.8	13.7	21.8	14.6	21.6	42.5
g721dec	160806.0	175845.0	175596.0	197398.0	162451.0	176087.0
unepic	7881.0	7553.0	8871.0	9640.0	8784.0	9543.0
mpeg2dec	82568.0	77626.0	82948.0	113452.0	97939.0	108624.0
GEOMEAN	170.7	177.7	200.1	194.3	189.2	209.8
% change	-19%	-15%	-5%	-7%	-9%	

Table VI. Cycles comparison on a VLIW with 32 registers and 8 FUs

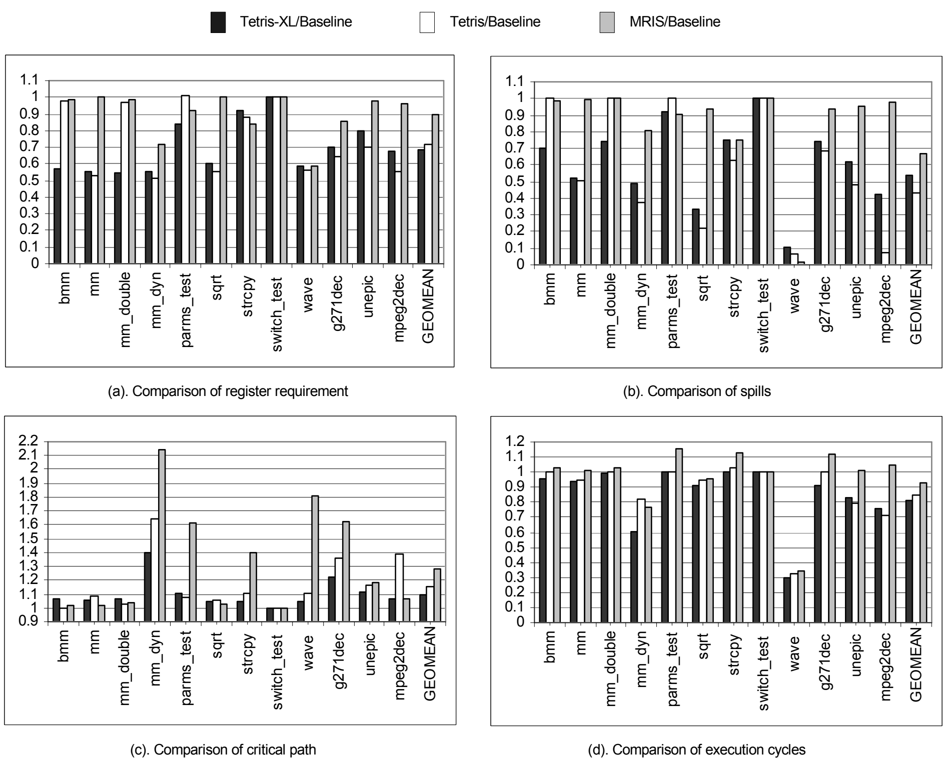

Fig. 20. Additional comparisons on a VLIW with 32 registers and 8 FUs

and 7%, respectively.

	Tetris-XL (flow 2)	Tetris (flow 3)	Greedy (flow 4)	MRIS (flow 5)	FBS (flow 6)	Baseline (flow 1)	
Benchmark	Spills (K)	Spills (K)	Spills (K)	Spills (K)	Spills (K)	Spills (K)	Spill ratio
bmm	0.3	0.3	0.3	0.3	0.3	0.3	1%
mm	1.7	1.7	1.8	1.7	1.8	1.8	1%
mm_double	1.7	1.7	1.7	1.7	1.7	1.7	1%
mm_dyn	1.7	1.7	1.7	1.7	1.7	1.7	1%
parms_test	1.7	1.8	1.8	1.8	1.7	1.8	16%
sqrt	0.1	0.1	0.2	0.2	0.2	0.2	5%
strcpy	0.0	0.0	0.1	0.0	0.1	0.1	1%
switch_test	2.5	2.6	2.6	2.6	2.6	2.6	14%
wave	0.1	0.1	0.1	0.1	0.1	0.1	1%
g721dec	43191.0	30840.0	40920.0	42896.0	41717.0	42896.0	10%
unepic	369.0	55.0	1632.0	1751.0	1443.0	1737.0	13%
mpeg2dec	5511.0	4043.0	5154.0	5691.0	5559.0	5680.0	3%
GEOMEAN	5.2	3.9	5.9	6.0	6.1	6.2	2%
% change	-16%	-37%	-6%	-3%	-2%		

Table VII. Spills comparison on a VLIW with 64 registers and 8 FUs

It is observed from Table V that Tetris achieves 11% more spill reduction on average than Tetris-XL. Tetris-XL is more conservative when register pressure is low because it evaluates both spill reduction and potential critical path increases. The aggressive spill reduction of Tetris comes at the cost of large critical path increase. As shown in Figure 20-c, the average critical path increases caused by Tetris-XL, Tetris and MRIS are 9%, 17% and 28%, respectively. Therefore, although Tetris-XL achieves less spill reduction compared with Tetris, it still outperforms Tetris and other techniques in terms of performance improvement.

Benchmark *wave* demonstrates the benefit of Tetris-XL. As shown in Figure 20-b,c, Tetris reduces spills of *wave* by 94% at the cost of a 11% increase in critical path length. Tetris-XL makes a tradeoff between spill reduction and critical path increase so that a smaller spill reduction of 89% is achieved with a smaller critical path increase of 5%. As a result of this tradeoff, Tetris-XL achieves an additional execution cycle reduction of 4% compared with Tetris, which is shown in Figure 20-d.

6.3 Experiments with trivial register pressure

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 [Transmeta, Inc. 2005] and the Texas Instruments C64x processor [Texas Instruments, Inc. 2000]. On average, spills only take up 2% of the total operations in the baseline flow in Table VII. Due to this trivial register pressure, Table VIII shows that Tetris-XL, Tetris and MRIS provides an average performance improvement of 2%, 1% and 1%, respectively. As expected, the benefit of register pressure control becomes marginal when register pressure is very low.

Benchmark	Tetris-XL (flow 2) Cycles (K)	Tetris (flow 3) Cycles (K)	Greedy (flow 4) Cycles (K)	MRIS (flow 5) Cycles (K)	FBS (flow 6) Cycles (K)	Baseline (flow 1) Cycles (K)
bmm	87.5	87.5	87.5	87.5	87.5	87.5
mm	92.6	93.7	92.6	92.6	92.6	92.6
mm_double	89.4	89.3	89.3	89.3	89.3	89.3
mm_dyn	29.9	29.2	29.9	32.0	29.9	29.9
parms_test	5.6	5.6	5.6	5.6	5.6	5.6
sqrt	3.8	3.8	3.8	3.8	3.8	3.8
strcpy	18.3	19.0	18.3	18.3	18.3	18.3
switch_test	13.0	13.0	13.0	13.0	13.0	13.0
wave	12.1	12.9	12.1	12.3	12.1	12.1
g721dec	162842.0	162225.0	165303.0	163373.0	162189.0	162785.0
unepic	5575.0	5880.0	7236.0	7151.0	6613.0	7125.0
mpeg2dec	74581.0	75318.0	74577.0	74773.0	74590.0	74673.0
GEOMEAN	145.3	147.5	148.7	149.6	147.6	148.4
% change	-2%	-1%	0%	0%	-1%	

Table VIII. Cycles comparison on a VLIW with 64 registers and 8 FUs

7. SUMMARY AND FUTURE WORK

In this paper, we present new spill reduction techniques to improve the performance of VLIW processors with limited registers. By modifying the relative ordering of operations, this technique serializes multiple variables simultaneously so that the register requirement can be reduced with a limited critical path increase. For VLIW programs that experience high register pressure, this technique reduces spills and improves execution time by identifying operations that are likely to create spills. For a 4-way VLIW architecture with 16 registers, our heuristic reduces the average execution time by an additional 14% compared with previous spill reduction techniques by simultaneously considering numerous variable serializations. For a 8-way VLIW architecture with 32 registers, the additional execution time reduction is 10%. For architectures with 64 registers, little additional execution time reduction is achieved.

Several areas of future work are apparent from this research. The effect of cache sizes and protocols, as well as other dynamic microarchitectural features, could be considered in the context of high register pressure. Additionally, more tradeoffs could be considered in Tetris to reduce compile time. Unlikely serializations could be pruned earlier in partitioning to reduce serialization search time. The number and type of functional units could be considered by an expanded version of Tetris-XL in an effort to avoid the optimization of infeasible schedules. Finally, a theoretical treatment of the conditions that lead to Max_reg reduction could be explored.

ACKNOWLEDGMENTS

The authors wish to acknowledge the efforts of Premachandran R. Menon and David Howland in the completion of this work.

REFERENCES

Berson, D. A., Gupta, R., and Soffa, M. L. 1993. 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*. 243–254.

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

Bouchez, F., Darte, A., and Rastello, F. 2007. On the Complexity of Spill Everywhere under SSA Form. In *ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*. 103–112.

Briggs, P. 1992. Register Allocation via Graph Coloring. Ph.D. thesis, Department of Computer Science, Rice University.

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

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

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

Cilio, A. and Corporaal, H. 1999. Global Program Optimization: Register Allocation of Static Scalar Objects. In *Conference of the Advanced School for Computing and Imaging*. 52–57.

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

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

Faraboschi, P., Brown, G., Fisher, J. A., Desoli, G., and Homewood, F. 2000. Lx: A Technology Platform for Customizable VLIW Embedded Processing. In *International Symposium on Microarchitecture*. 203–213.

Freescale Semiconductor, Inc. 2005. *MSC8101 Reference Manual*. Freescale Semiconductor, Inc.

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

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

Goossens, G., Praet, J. V., Lanneer, D., and Geurts, W. 1997. Embedded Software in Real-Time Signal Processing Systems: Design Technologies. *Proceedings of the IEEE 85,* 3 (Mar.), 436–454.

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

Hennessy, J. L. and Patterson, D. A. 1996. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publisher.

Kim, H. 2001. Region-based Register Allocation for EPIC Architectures. Ph.D. thesis, Department of Computer Science, New York University.

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

Marquardt, A., Betz, V., and Rose, J. 2000. Timing-driven placement for FPGAs. In *ACM International Symposium on Field Programmable Gate Arrays*. 203–213.

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

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

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

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

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

Transmeta, Inc. 2005. *Transmeta Efficeon TM8820 Processor*. Transmeta, Inc.

Xu, W. and Tessier, R. 2007. Tetris: A New Register Pressure Control Technique for VLIW Processors. In *ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems*.

Zeitlhofer, T. and Wess, B. 2003. List-coloring of interval graphs with application to register assignment for heterogeneous register-set architectures. *Signal Processing 83,* 7 (July), 1411–1425.

A. APPENDIX

Lemma 1. $Succ(u) \cap Desce(v) \neq \emptyset \Leftrightarrow$ Serialization $(u \rightarrow v)$ creates a cycle.

Sufficiency proof: $Succ(u) \cap Desce(v) \neq \emptyset \Rightarrow \exists$ a path (v to w), where $w \in (Succ(u) - v)$. Serialization $(u \rightarrow v) \Rightarrow$ set of serial edges: (p to v), where $p \in (Succ(u) - v) \Rightarrow \exists$ a serial edge (w to v). Therefore, a path (v to w) and a serial edge (w to v) create a cycle between w and v.
Necessity proof: Serialization $(u \rightarrow v)$ creates a cycle $\Rightarrow$ the cycle must include at least one serial edge (w to v), where $w \in (Succ(u) - v) \Rightarrow \exists$ a path (v to w), where $w \in Succ(u) - v$. Therefore, $Succ(u) \cap Desce(v) \supseteq \{w\} \neq \emptyset$.

Lemma 2. Serializations $(u \rightarrow v)$ & $(s \rightarrow t)$ create a cycle $\Leftrightarrow$ ($v \in Succ(s)$ or $Succ(s) \cap Desce(v) \neq \emptyset$) & ($t \in Succ(u)$ or $Succ(u) \cap Desce(t) \neq \emptyset$).

Sufficiency proof. If $(u \rightarrow v)$ & $(s \rightarrow t)$ create a cycle, the cycle must include a serial edge (w to v), where $w \in Succ(u)$ and a serial edge (x to t), where $x \in Succ(s)$ $\Rightarrow$ the cycle can be represented as (w to v) to (x to t) to $w \Rightarrow$ there must be a connection $A =$ (v to x) & a connection $B =$ (t to w).

Connection $A =$ (v to x) $\Rightarrow v = x$ or $\exists$ path (v to x) $\Rightarrow v \in Succ(s)$ or $Succ(s) \cap Desce(v) \supseteq \{x\} \neq \emptyset$.

Connection $B =$ (t to w) $\Rightarrow t = w$ or $\exists$ path (t to w) $\Rightarrow t \in Succ(u)$ or $Succ(u) \cap Desce(t) \supseteq \{w\} \neq \emptyset$.

Therefore, $(u \rightarrow v)$ & $(s \rightarrow t)$ create a cycle $\Rightarrow$ ($v \in Succ(s)$ or $Succ(s) \cap Desce(v) \neq \emptyset$) & ($t \in Succ(u)$ or $Succ(u) \cap Desce(t) \neq \emptyset$).

Necessity proof. ($v \in Succ(s)$ or $Succ(s) \cap Desce(v) \neq \emptyset$) & ($t \in Succ(u)$ or $Succ(u) \cap Desce(t) \neq \emptyset$) $\Rightarrow$

(a) $v \in Succ(s)$. Serialization $(s \rightarrow t) \Rightarrow$ set of serial edges:$\{$(p to t), where $p \in (Succ(s) - t)\} \Rightarrow \exists$ a serial edge (v to t).

(b) $Succ(s) \cap Desce(v) \neq \emptyset) \Rightarrow \exists$ path (v to x), where $x \in Succ(s)$. Serialization $(s \rightarrow t) \Rightarrow$ set of serial edges:$\{$(p to t), where $p \in (Succ(s) - t)\} \Rightarrow \exists$ a path (v to t) = path (v to x) + serial edge (x to t), where $x \neq t$ or $\exists$ a path (v to t) = path

(v to x), where $x = t$;

(c) $t \in Succ(u)$. Serialization $(u \rightarrow v) \Rightarrow$ set of serial edges:(q to v), where $q \in (Succ(u) - v) \Rightarrow \exists$ a serial edge (t to v).

(d) $Succ(u) \cap Desce(t) \neq \emptyset \Rightarrow \exists$ path (t to w), where $w \in Succ(s)$. Serialization $(u \rightarrow v) \Rightarrow$ set of serial edges:(q to v), where $q \in (Succ(u) - v) \Rightarrow \exists$ a path (t to v) = path (t to w) + serial edge (w to v), where $w \neq v$ or $\exists$ a path (t to v) = path (t to w), where $w = v$.

Therefore, with (a) or (b), $(s \rightarrow t) \Rightarrow \exists$ a serial edge (v to t) or $\exists$ a path (v to t); with (c) or (d), $(u \rightarrow v) \Rightarrow$ a serial edge (t to v) or $\exists$ a path (t to v); Finally, ((a) or (b)) & ((c) or (d)) = ($v \in Succ(s)$ or $Succ(s) \cap Desce(v) \neq \emptyset$) & ($t \in Succ(u)$ or $Succ(u) \cap Desce(t) \neq \emptyset$) $\Rightarrow$ Serializations $(s \rightarrow t)$ and $(u \rightarrow v)$ create a cycle (v, t).