HEURISTIC TRADEOFFS BETWEEN LATENCY AND ENERGY CONSUMPTION IN REGISTER ASSIGNMENT *

R. ANAND, M. JACOME, AND G. DE VECIANA

Department of Electrical and Computer Engineering
University of Texas at Austin, Austin, TX 78712

Tel: (512) 471-2051 Fax: (512) 471-5532

{randy,jacome,gustavo}@ece.utexas.edu

Abstract

One of the challenging tasks in code generation for embedded systems is register allocation and assignment, wherein one decides on the placement and lifetimes of variables in registers. When there are more live variables than registers, some variables need to be *spilled* to memory and restored later. In this paper we propose a policy that minimizes the number of spills – which is critical for portable embedded systems since it leads to a decrease in energy consumption. We argue however, that schedules with a minimal number of spills do not necessarily have minimum latency. Accordingly, we propose a class of policies that explore tradeoffs between assignments leading to schedules with low latency versus those leading to low energy consumption and show how to tune them to particular datapath characteristics. Based on experimental results we propose a criterion to select a register assignment policy that for 99% of the cases we considered minimizes both latency and energy consumption associated with spills to memory.

1 Introduction

Embedded processor cores used in today's embedded systems place heavy burdens on current compiler technology. A number of difficulties stem from architectural specializations in embedded processors [9, 10]. In this paper we focus on *clustered* VLIW ASIPs which are well suited to increasingly pervasive (portable) embedded multimedia/communications applications. A clustered ASIP has a distributed set of register files each connected to a dedicated set of functional units, e.g., Fig.2. Such an organization can significantly reduce the area/delay/power cost of storage and communication [4] but, if not properly accounted for during code generation, can result in degraded performance [5, 9, 10].

A number of researchers have suggested that the first phase in code generation for such clustered machines should be the binding of operations and variables to the datapath's clusters [5, 6, 11]. In order to avoid penalties associated with data transfers, a key objective in performing cluster assignment is to try to keep operations that *share* variables on the same cluster while still maximizing instruction level parallelism. However, since local storage resources have finite capacity some variable sharing opportunities may be infeasible. Indeed, when register files fill up, variables may need to be spilled to, and recovered from, memory. This not only increases energy consumption but typically also increases latency. The focus of this paper is on determining variable replacement policies, i.e., register assignment policies, that avoid such overloads and explore tradeoffs to achieve low latency and energy consumption.

*This work is supported by a National Science Foundation NSF Career Award MIP-9624231, NSF Award CCR-9901255 and Grant ATP-003658-0649-1999 of the Texas Higher Education Coordinating Board.

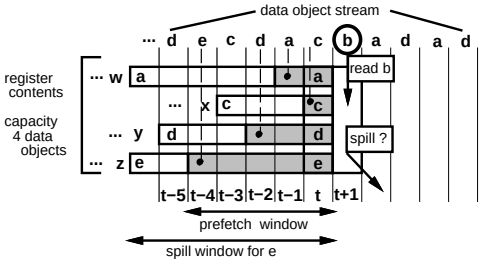

Figure 1: Replacement policies, spilling and prefetching windows.

The key ideas in this paper can be summarized based on the simple example shown in Fig.1. The figure exhibits the state of a single register file (of size 4) up to time t and a stream of variables that it needs to support, i.e., the variables that must be in the register at each step. The stream is shown at the top of the figure and for simplicity contains only one variable per step. At time t the register contains $\{a, c, d, e\}$, and the variable b must be loaded at time $t+1$. The basic question is: which of the variables currently in the register file should b replace, i.e., what criterion should be used in selecting which variable to spill ?

A *forward looking policy* chooses to replace variables whose next use is the furthest away. Thus, for example, since neither c nor e appear in the data stream after time t, they are good candidates for spilling. In §3.1 we show that such a policy minimizes the overall number of variable replacements. By minimizing the number of replacements we not only reduce energy consumption, but also maximize the *average* contiguous lifetimes of variables in the register file. We call such intervals *spilling windows* since they correspond to intervals over which one could choose to perform a spill to memory without necessarily delaying the schedule. If we choose to replace e at time $t+1$ then looking back we note that its spilling window would have been quite long, e.g., 7 time steps.

Alternatively, a *backward looking policy* might choose to replace the least recently used variable, i.e., the well known LRU policy. In §3.2 we show that this policy generates large *prefetching windows*. The prefetching window associated with a new variable entering the register represents a window of opportunity during which it can be pre-loaded from memory, without introducing further delays. In Fig.1 the prefetching windows associated with possible replacement choices are shown in gray. Thus, for example, if we choose to replace variable c with b, a prefetching window of length 1 would be obtained since c was used on the previous step. By contrast if we chose to replace variable e with b a prefetching window of length 5 would be obtained. Clearly, from the point of

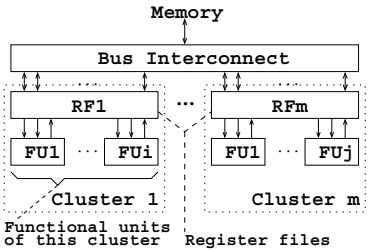

Figure 2: Illustration of a simple datapath.

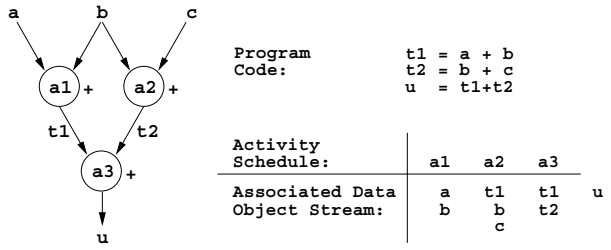

Figure 3: Data stream for a given binding/schedule of activities.

view of maximizing the interval of time available for prefetching the new variable *b*, *e* is a better choice to spill.

The overall latency of a schedule will depend on which variables are spilled and the sizes of both their spilling and prefetching windows as well as the latency of writing/reading to/from memory. Thus in §3.3 we propose a family of heuristic policies that explore tradeoffs between reducing the number of spills (and thus maximizing the average spilling windows) and achieving large prefetching windows. As will be discussed in the sequel, the proposed *tradeoff policy* is a basic tool to attempt to get a handle on our problem, i.e., to find a way to 'steer' data objects to and from memory that results in a schedule with low latency and energy consumption.

1.1 Related work

Graph coloring based approaches are commonly adopted to perform register allocation. The idea is to determine the number of colors (registers) required to cover the interval graph associated with the lifetimes of variables [2]. Modified forms of this algorithm are used in the FlexCC compiler [9], the ROCKET Compiler [12] and the AVIV retargetable code generator [5]. A version of the graph coloring approach, called the Left Edge Algorithm, is a greedy algorithm which explicitly determines a register assignment requiring a minimum number of registers [1, 8, 9]. Specifically, it starts by sorting the variables in increasing order of birth. Then, starting from those with earliest birth, it assigns those with non-overlapping lifetimes to the first register. If a sub-set of variables still remains unassigned, a new register is created and the process is repeated on the remaining variables. Variants of this algorithm have been used in a number of compilers, e.g., CodeSyn [3]. Unfortunately this method does not exploit locality when variables have non-trivial (i.e., non-contiguous) lifetimes in order to reduce spills for fixed size register files.

Kolson et. al., [7] proposed an optimal, though exponential time, algorithm which assigns variables to registers so as to minimize the number of spills. They also report a heuristic with a polynomial run time that gives good results. However as will be seen in the sequel minimization of spills need not translate to a minimum latency schedule.

1.2 Paper organization

In §2 we introduce notation and discuss the problem setup. In §3 we analyze the forward, backward and tradeoff replacement policies in the context of a *single* cluster. In §4 we discuss heuristics that exploit more detailed information on the dataflow's variables characteristics. We briefly discuss our approach for datapaths with multiple clusters in §5. Experimental results and conclusions are included in §6 and §7 respectively.

2 Problem formulation

Datapath and dataflow model. The target family of VLIW ASIPs consists of *storage resources*, *functional units* and a *bus interconnect* as shown in Fig.2. Storage resources are of two types: finite capacity register files and "infinite" capacity memory blocks. Functional units are connected to register files from which they draw their operands and place their results. We assume that primary inputs required for execution are loaded from memory into the register files using the finite capacity bus interconnect. Similarly, primary outputs generated during execution are stored into memory through the bus.

A dataflow is modeled by a polar DAG $G(A,T)$ where A is the set of *activities* (operations) to be executed and the edges T are labeled with data objects corresponding to the program's variables, and its primary inputs and outputs. The edges represent both precedence constraints among activities and data transfers that may be necessary to bring data objects from producer activities to consumer activities. Data objects are further partitioned into three disjoint sets $D = PI \cup PO \cup LD$ corresponding to: primary inputs which are initially stored in memory, primary outputs which must be output to memory, and local data objects which are generated and consumed internally but need not be output to memory. Primary inputs/outputs are associated with edges exiting/abutting in the source/sink node of the polar DAG. We let ID_a denote the set of input data objects for an activity $a \in A$ and RD_a denote a set with the resulting data object.

Problem statement. As discussed in the introduction, for clustered machines the binding of activities and data objects to clusters is a critical step that should be performed early on in code generation [5, 6, 11]. In this paper we assume that such a binding has been determined and we are given a partial order for the activities' execution $\vec{S} = (S_t | 0 \le t \le T-1)$ where $S_t \subset A$ is a set of activities to be executed prior to those in S_{t+1}. This partial order results from the coarse/simplified scheduling problems used to drive the cluster binding phase, that ignore some datapath specifics, e.g., register capacities and data transfers [6].

For simplicity, we first consider a datapath with a single register file, i.e., single cluster and discuss extensions to datapaths with multiple clusters in §5. Suppose an activity $a \in S_t$ is scheduled on step t. Then its operand(s) ID_a, must be present in the register file at time t and the result RD_a must be placed in the register file at step $t+1$. Thus we can translate $\vec{S}$ into a sequence of data objects that must be supported by the register file over time, i.e., $\vec{D} = (D_t | 1 \le t \le T)$ where $D_t \subset D$ is given by

$$D_t = (\bigcup_{a \in S_t} ID_a) \bigcup (\bigcup_{a \in S_{t-1}} RD_a) \quad \text{for} \quad 1 \le t \le T.$$

We shall assume that $|D_t| \le R$ where R denotes the size of the register file.

We let $X_t \subset D$ denote the set of data objects in the register file at time t, where $|X_t| \le R$. In order to ensure that $D_t \subset X_t$ for all t, data transfers may need to be scheduled, possibly delaying execution of the activities in S_t. Given these requirements, we can consider various ways of 'steering' data objects between the register file and the memory banks so that activities can be executed as soon as possible but in the proposed order. Consider the dataflow shown in Fig.3, and suppose that all operations are bound to a single ALU connected to a register file. The figure shows the resultant data object stream for the given partial order for the activities.

Our problem is to find an optimal way to 'steer' data objects to and from memory that will result in a schedule with low latency and energy consumption.

3 Replacement policies - Spilling and prefetching windows

We shall consider various data object replacement policies paying special attention on the resulting spilling and prefetching windows. All policies are parameterized based on three simple control actions, 'load,' 'replace' and 'do nothing'. A load(b) action corresponds to loading an additional data object $b \in D$ into the register file, which is admissible only if there is free space in the file. The replace(a,b) action corresponds to replacing a data object a, currently in the register file, with data object b.

3.1 Forward policy

Let $\vec{D} = \{D_t | 1 \le t \le T\}$ represent a register's data stream, where D_t is the set of data objects that need to be in the register file at time t. Our goal is to select a sequence of control actions that ensure that $D_t \subset X_t$ for each time step t. In general, control actions are parameterized by pairs of sets of data objects (A,B) where $A,B \subset D$. Such pairs are interpreted as replace(A,B) i.e., the action of replacing the data objects in A with those in B. For example, if $A = \{a,b\}$ and $B = \{c,d,e\}$ then a,b would be replaced with c,d,e. Clearly these correspond to a set (not necessarily unique) of load and replace actions, e.g., $\{\text{replace}(a,c), \text{replace}(b,d), \text{load}(e)\}$. Given these control choices the dynamics of the register file contents can be described as follows:

Admissible action space: let $U(X_s, D_{s+1})$ denote the set of admissible actions at time s when the register contents are X_s. An action $U_s = (A,B) \in U(X_s, D_{s+1})$ is admissible if it results in a new register state X_{s+1} satisfying $D_{s+1} \subset X_{s+1}$ and $|X_{s+1}| \le R$. To be admissible an action $U_s = (A,B)$ must be such that $A \subset X_s, B \cap X_s = \emptyset$, and $|A| \le |B|$.

System dynamics: let f denote the system dynamics corresponding to modifying the contents of the register bank according to an admissible action $U_s = (A,B)$ which replaces A with B, i.e., $X_{s+1} = f(X_s, U_s) = (X_s \setminus A) \cup B$.

Cost of an action: we assume the cost, $c(U_s)$, of an action $U_s = (A,B)$ is given by $|B|$ the total number of data objects loaded into the register.

Next we define the problem associated with determining a replacement policy with minimum cost, i.e., resulting in a minimum overall number of loads, and an algorithmic for this problem.

Problem 1 *Given a register bank of size R, with initial state X_0, that needs to support the data sequence $\vec{D}$ find a sequence of controls $\vec{U} = (U_s | 0 \le s \le T-1)$ with minimum overall cost:*

$$J^*(X_0, \vec{D}) = \min_{\vec{U}} \{ \sum_{s=0}^{T-1} c(U_s) | X_{s+1} = f(X_s, U_s), U_s \in U(X_s, D_{s+1}) \}.$$

Algorithm 3.1 (Forward Policy) *The following policy is optimal for Problem 1. Starting from $t = 0$ with initial state X_0 proceed forward until $T-1$. At time t, given the state of the register bank X_t, let $B = D_{t+1} \setminus X_t$ and select actions as follows:*

- *if $B = \emptyset$, do nothing;*
- *else replace(A^*, B) where $A^* \subset X_t \setminus D_{t+1}$ is a (not necessarily unique) set of $\max[0, |X_t| + |B| - R]$ data objects with the largest $l_t(a)$, where $l_t(a)$ is given by*

$$l_t(a) = \min\{T-1, \ \min[s \mid a \in D_s \text{ and } t < s \le T] \ \}.$$

The forward policy corresponds to replacing data objects only when necessary, and replacing those objects which will be used the latest (or not used again) first, i.e., those with the largest $l_t()$. Space precludes us from presenting our proof of optimality – it is based on dynamic programming results.

X_0	X_1	X_2	X_3	X_4	X_5	X_6	X_7
a	a	a	a	x	c	c	a
y	y	b	b	b	b	b	b
$\vec{d}$	a	b	b	x	c	b	a
X_0	X_1	X_2	X_3	X_4	X_5	X_6	X_7
a	a	$\rightarrow$	$\rightarrow$	x	c	$\rightarrow$	a
y	$\rightarrow$	b	$\rightarrow$	$\rightarrow$	$\rightarrow$	$\rightarrow$	$\rightarrow$
$\vec{d}$	a	b	b	x	c	b	a

Table 1: Forward policy: state evolution, spilling windows.

This policy minimizes the number of state changes (cost) by keeping data objects which are likely to be used in the sequel in the register file. As a consequence it also maximizes the *average* length of spilling windows. Table 1 exhibits the state evolution of the register file for the forward policy using an example. The table exhibits the data stream $\vec{D}$ and the states of the register file X_t. Consider the first row of data objects in the register. It shows that a, which is needed in the register at time 1, is replaced by x at time 4. We call this time interval its *spilling window* and denote its length by spillwin$(a,x) = 4-1 = 3$. As discussed in the introduction large spilling windows correspond to available time to make a possible spill of a to memory combined with a load of x into the register file. The table below shows such spilling windows using right arrows ($\rightarrow$) to indicate that a transaction can take place during this time interval. Note that data object b first appears in the register at time 2 and can be written to memory (if needed) thereafter. By contrast, x is immediately replaced with c and has a spilling window of length 1. So there is little leeway for spilling x to memory before time 5. We may expect the schedule to be delayed if accesses to memory are lengthy. Thus, although the number of state changes is a minimum and the *average* size of the spill windows is large, we may have some data objects with very large spill windows that are not fully utilized and others with very small spill windows that force the delaying of the schedule. This suggests that it may be desirable to explore alternate policies that would generate spill windows that are consistently large.

3.2 Backward policy

Suppose $b \in D_t$ is a data object that needs to be in the register file at time t. If the register is not currently full, we can simply load the data object – in fact we could have prefetched it earlier. However if the register is full, a data object currently in the register file will need to be replaced. We will consider a greedy policy which looks back in time and selects a data object $a^* \in X_t$ that was *least recently used*, i.e., $a^* \in \text{argmin}_a \{p_t(a) | a \in X_t\}$ where $p_t(a)$ denotes the time that a was last used or is set to 0, i.e.,

$$p_t(a) = \max\{0, \ \max[s \mid a \in D_s \text{ and } 1 \le s < t] \ \}.$$

This backward looking policy is summarized below.

Algorithm 3.2 (Backward Policy) *Starting from $t = 0$ with initial state X_0 proceed forward to $T-1$. At time t, given the state of the register bank X_t, let $B = D_{t+1} \setminus X_t$ and select actions as follows:*

- *if $B = \emptyset$, do nothing;*
- *else replace(A^*, B) where $A^* \subset X_t \setminus D_{t+1}$ is a (not necessarily unique) set of $\max[0, |X_t| + |B| - R]$ data objects with the smallest $p_t(a)$.*

Suppose a data object a^* is replaced by b in the register file at time t, then the prefetching window is given by prefetchwin$(a^*, b) = t - p_t(a^*)$. This measures the time frame during which the data object b could have been loaded from memory. Note that if b were the result of an activity at step $t-1$, one would not be able to prefetch the data object. For the time being we ignore such information which is of course embedded in the dataflow. In the case where a data object is loaded at time t without replacement we shall denote its prefetching window by prefetchwin$(\emptyset, b) = t$.

X_0	X_1	X_2	X_3	X_4	X_5	X_6	X_7
a	a	a	a	x	x	b	b
y	y	b	b	b	c	c	a
$\vec{d}$	a	b	b	x	c	b	a
X_0	X_1	X_2	X_3	X_4	X_5	X_6	X_7
a	a	$\rightarrow$	$\rightarrow$	x	$\rightarrow$	b	b
y	$\rightarrow$	b	b	$\rightarrow$	c	$\rightarrow$	a
$\vec{d}$	a	b	b	x	c	b	a

Table 2: Backward policy: state evolution, prefetching windows.

Fact 3.1 *Suppose the "backward policy" is used to determine a sequence of controls for the register bank to support the data sequence $\vec{D}$ given an initial condition X_0. Assume that $\vec{D}$ satisfies $|D_t| \leq M$ for all t, and suppose (for simplicity) that $R = kM$ for some integer k. The prefetching window associated with any replacement of a data object in the register file, say replace(a^*, b), at any time $t \geq k$ satisfies*

$$\text{prefetchwin}(a^*, b) = t - p_t(a^*) \geq k - 1.$$

Similarly if the register is not full at time t and an object b is loaded its prefetching window is given by $\text{prefetchwin}(\emptyset, b) = t \geq k - 1$.

Due to space constraints we have not included a proof of Fact 3.1. Its significance is that it assures us that the prefetching windows associated with the backward policy will eventually always exceed some minimal size. Such uniformity, has advantages as it ensures all replacements will have a reasonable time to take place. The backward policy, however, has some drawbacks of its own. It may incur many more changes in state, and since the capacity of the bus interconnect is limited it may result in increased delays. The increased number of fetches and spills may increase the latency of the schedule although one has consistently 'large' prefetching windows in which to load data objects. Table 2 shows the state evolution obtained for our simple example using the backward policy as well as the associated prefetching windows. Note that all prefetching windows exceed (R/M), viz., 2. The number of state changes is 5 with the backward policy while it was 4 with the forward policy. As we will see in §6 such comparisons are more interesting when we quantify the latency of a schedule for a given datapath and use it as a measure to rank the performance of a policy.

3.3 Tradeoff policy - Latency versus Energy Consumption

To find a compromise between minimizing state changes (and thus energy consumption) and obtaining large prefetching windows we propose to use policies that look both forward and backward. Suppose data object $a \in X_t \setminus D_{t+1}$ is a candidate for replacement at time t. The tradeoff policy proposed below takes decisions on replacement based on a ranking function $r_t(a)$ that may depend on both $p_t(a)$ and $l_t(a)$ as well as various other aspects of the problem.

Algorithm 3.3 (Tradeoff Policy) *Starting from $t = 0$ with initial state X_0 proceed forward to $T - 1$. At time t, given the state of the register bank X_t, let $B = D_{t+1} \setminus X_t$ and select actions as follows:*

- *if $B = \emptyset$, do nothing;*
- *else replace(A^*, B) where $A^* \subset X_t \setminus D_{t+1}$ is a (not necessarily unique) set of $\max[0, |X_t| + |B| - R]$ data objects with the largest ranks $r_t(a)$, where ties are broken arbitrarily.*

Note that the backward and forward policies are special cases with ranking functions given by $-p_t(a)$ and $l_t(a)$ respectively. In looking backward, it is desirable to replace data objects that have been in the register file for a long time, since such replacements will be associated with large transaction windows. In looking forward it is desirable to retain data objects that might be reused in the *near future*. We might however want to preempt the formation of large spill windows, particularly when the number of registers is low. To achieve tradeoffs between these two policies we considered (among others) the following ranking function:

$$r_t(a) = \alpha(l_t(a) - t) + \beta(t - p_t(a)) \quad \text{where} \quad \alpha, \beta \geq 0,$$

are parameters that emphasize the minimization of state changes and sizes of prefetching windows respectively.

4 Heuristics that account for data object type

We propose data type dependent *tie breaking* heuristics to further improve our register assignment policies. They are based on the characterization of data objects, as PI, PO or LD, and the history of the data object in the schedule. We propose the following policies:

1. If one of the data objects is not used in the future, it is a better candidate for replacement.
2. PIs and POs are preferred over LDs for replacement because there is only one memory transaction associated with them. An LD has to be stored and then retrieved when needed.
3. LDs that have once been spilled to memory are treated as PIs. Future replacements of this LD do not require a write to memory.

5 Clustered VLIW datapaths

Until now for simplicity we have focused on a single register file. Our approach can however be applied to clustered datapaths such as that shown in Fig.2. Note that, the same data object may now be required in different register files. As discussed in the sequel, this does not affect our policies in any significant way. In this new scenario we have m data streams corresponding to m register files which we assign using our policies as before. We do, however, need to enforce data-dependency constraints across clusters. Specifically, we need to ensure that a data object is not read from a register file by a functional unit before it has been created in some other register file and copied to its current location. In such cases one can introduce stalls in the schedule on a given register file while waiting for the creation of a data object on another.

6 Experimental Results

Until here we have discussed a class of policies aimed at deciding which data objects to spill to memory when required, with a view on reducing both latency and number of load/stores to memory (energy consumption). Based on a replacement sequence obtained using such a policy, we propose to greedily schedule data transfers to ensure that activities are executed in the specified order, see §2. This greedy schedule makes the most of spill and prefetch windows, and enforces if need be, synchronization constraints among streams assigned to different register files, see §5. Space precludes us from giving a detailed account of this process. Although the obtained schedule may not be optimal it certainly exploits the locality of the stream, the spill and prefetch windows, as well as the available buses in a greedy fashion.

In our examples we considered the class of policies discussed in §3.3 which are parameterized by $\alpha \in [0, 1]$ where $\alpha + \beta = 1$. Thus $\alpha = 0$ corresponds to the backward policy and $\alpha = 1$ to the forward policy. We consider register assignment policies for loop bodies of an FFT and a 4th order Avenhous filter bound to a clustered datapath as shown in Fig.2. We generated results for a total of 144 cases associated with the FFT and Avenhous filter and several datapaths with varying register file sizes, load/store latencies and bus widths.

In 99% of our cases, the schedules obtained from replacement policies having $\alpha \in [0.6, 0.8]$ resulted in both minimum latency and read/writes to memory. Note that α in this range roughly corresponds to a mix of the forward and backward policies that places more weight on the forward view. Although one might expect the

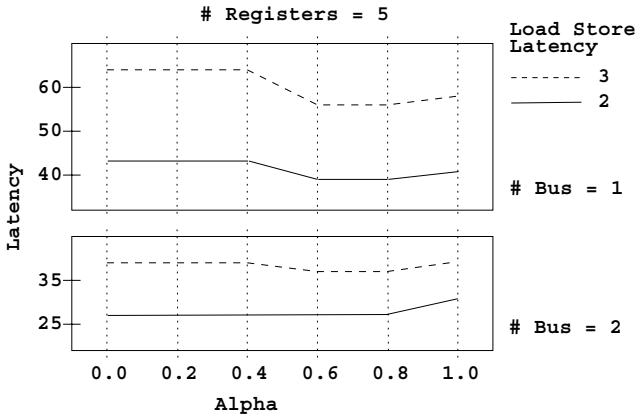

Figure 4: Latency obtained for different α for datapaths with different number of buses and load/store latencies.

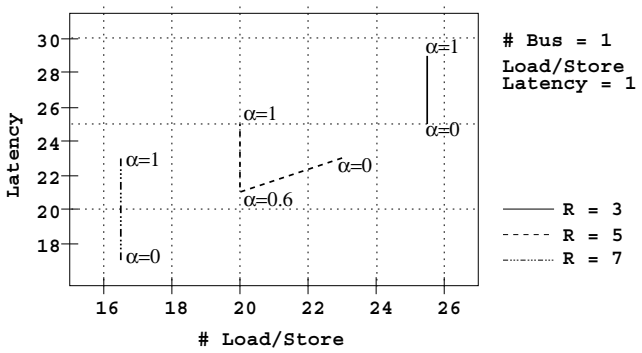

Figure 5: Latency vs. number of load/stores for different register file sizes and different α.

forward policy to minimize spills and thus also latency, our results show that slightly biasing it with the backward view, and thus increasing prefetch windows, results in even lower latency. The results shown in Fig. 4 are typical of the cases we considered. The figure exhibits the latency obtained as α was varied from 0 to 1, for a cluster with register file of size 5, bus width 1 and 2 and load/store latencies of 2 and 3. As can be seen, the range for α indicated above corresponds to the best latency in these four cases.

Next we briefly consider how register file size impacts the effectiveness of the proposed policies. Fig.5 exhibits the latency and number of load/stores obtained for various register files sizes (R= 3,5 and 7) as the parameter α was increased from 0 to 1. As expected, decreasing the register file size increases latency and the number of load/stores. Perhaps more interesting was a systematic trend we observed with respect to the two extremes associated with "large" and "small" register files. Indeed when the register file size is of the order of the number of live variables, i.e., large, the schedules obtained for different α have precisely the same number of load/stores, i.e., all the primary inputs/outputs that must be loaded/stored from/to memory. Note that further increase in the register file size can not decrease the number of load/stores. Similarly, when the register file size is of the order of the number of new variables required at each time step, i.e., small, different α have once again little impact on the resulting number of load/stores. Indeed, in this scenario, there are few opportunities to keep local data objects and primary inputs in the register file so that they may be reused later and thus the same number of load/stores are obtained for the various α. Finally when the register file is medium sized, we notice a marked change in the number of load/stores as α varies. As mentioned above, although the forward policy results in minimal load/stores, when the forward policy is slightly biased with the backward view one can achieve both minimum latency and minimum load/stores.

In general we found that examples with data streams in which variables appeared repeatedly and datapaths with sufficiently large register files, i.e., enabling reuse, the choice of α had a more significant effect on latency and number of load/stores. Nevertheless α within the range $[0.6, 0.8]$ consistently gave the best results over a wide range of datapaths for our two dataflow examples.

7 Conclusions

We discuss a novel approach to the register assignment problem aimed at both exploiting the locality in the streams of data to be supported by the register files as well as incurring low delays due to spills to memory. We propose a parameterized class of data object replacement policies that covers various compromises that might need to be made when determining a minimum latency schedule for a dataflow as well as minimization of spills due to energy consumption on a given datapath. Our experimental studies show that the policies enable one to explore these compromises in a systematic fashion. This work is complementary to the work in [6] which strives to find good joint binding/scheduling of a dataflow to clustered datapaths so as to minimize the required data transfers among register files.

References

[1] G. de Micheli. *Synthesis and Optimization of Digital Ciruits*. McGraw-Hill, Inc, 1994.

[2] G. Chaitin et al. Register allocation via coloring. *Computer Languages*, 6:47–57, Jan. 1981.

[3] P. Paulin et al. FlexWare: A flexible firmware development environment for embedded systems. In *Code Generation for Embedded Processors*, pages 67–84. KAP, 1995.

[4] S. Rixner et. al. Register organization for media processing. In *6th International Symposium on High-Performance Computer Architecture*, 2000.

[5] S. Hanono and S. Devadas. Instruction selection, resource allocation, and scheduling in the aviv retargetable code generator. In *Proceedings of the 35th Design Automation Conference*, 1998.

[6] M. Jacome and G. de Veciana. Lower bound on latency for VLIW ASIP datapaths. In *IEEE/ACM International Conference on Computer Aided Design*, 1999.

[7] D. Kolson, A. Nicolau, N. Dutt, and K. Kennedy. Optimal register assignment to loops for embedded code generation. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 1996.

[8] F. Kurdahi and A. Parker. REAL: A program for REegister ALlocation. In *Proc. of the 24th DAC*, pages 210–15, 1987.

[9] C. Liem. *Retargetable compilers for embedded core processors*. Kluwer Academic Publishers, 1997.

[10] P. Marwedel and G. Goossens, editors. *Code Generation for Embedded Processors*. Kluwer Academic Publishers, 1995.

[11] B. R. Rau, V. Kathail, and S. Aditya. Machine-description driven compilers for epic processors. Technical report, Hewlett-Packard Laboratories, 1998.

[12] P. Sweany and S. Beaty. Post-compaction register assignment in a retargetable compiler. In *Proceedings of the 23rd Annual Workshop in Microprogramming and Microarchitecture (MICRO-23)*, 1990.