Register Bank Assignment for Spatially Partitioned Processors

Behnam Robatmili, Katherine Coons, Doug Burger, and Kathryn S. McKinley

Department of Computer Science, The University of Texas at Austin
{beroy,coonske,dburger,mckinely}@cs.utexas.edu

Abstract. Demand for instruction level parallelism calls for increasing register bandwidth without increasing the number of register ports. Emerging architectures address this need by partitioning registers into multiple distributed banks, which offers a technology scalable substrate but a challenging compilation target. This paper introduces a register allocator for spatially partitioned architectures. The allocator performs bank assignment together with allocation. It minimizes spill code and optimizes bank selection based on a priority function. This algorithm is unique because it must reason about multiple competing resource constraints and dependencies exposed by these architectures. We demonstrate an algorithm that uses critical path estimation, delays from registers to consuming functional units, and hardware resource constraints. We evaluate the algorithm on TRIPS, a functional, partitioned, tiled processor with register banks distributed on top of a 4×4 grid of ALUs. These results show that the priority banking algorithm implements a number of policies that improve performance, performance is sensitive to bank assignment, and the compiler manages this resource well.

1 Introduction

Traditional architectures offer a single register file with uniform delay for reading from and writing to any architectural (physical) register. Register allocation assigns variables (virtual registers) to architectural registers when possible. In traditional graph coloring [4] and linear scan [14] algorithms, the only goal is to minimize the overhead of load and store instructions created by spilling variables to memory.

To address current technology scaling challenges, emerging architectures partition resources such as registers, caches, and ALUs. This approach provides the bandwidth needed for high ILP programs while increasing the resources available on the chip. Spatially partitioned processors have non-uniform register access times, which place more burden on the register allocator, requiring a more sophisticated algorithm that intertwines bank and register assignment. To optimize for the partitioned layout of these processors, the register allocator must consider the location of register banks and data caches, and the placement of instructions on ALUs in order to decide which register bank to use for each register. The delay in reading or writing a register depends on the length of the path

between the register bank and the ALU that reads or writes the register. Minimizing the communication latencies between partitioned components requires changes to traditional register allocation heuristics.

This paper proposes a bank allocation method for spatially partitioned architectures and evaluates it on the TRIPS hardware. The algorithm uses an evaluation function that selects a bank such that register values arriving at the same instruction at the same time are close to each other. The evaluation function calculates a score for each bank based on previously assigned banks of dependent instructions and the processor's topological characteristics.

We customize this evaluation function for TRIPS, a spatially partitioned tiled processor with register banks distributed in a row above a 4×4 array of ALUs. TRIPS is an instantiation of EDGE ISA in which each instruction encodes its consumer instructions directly. This direct, data-flow communication among instructions eliminates the need for an operand bypass network. Another characteristic of EDGE ISAs is atomic block execution, which amortizes the overhead of branch prediction and instruction cache access over a group of instructions. The results show that significant swings in performance are possible, and the algorithm improves over bank oblivious allocation by an average of 6%.

2 Related Work

Conventional register allocation methods. Chaitain et al. [4] present a graph-coloring register allocator that uses an interference graph (IG) to encode overlap between live ranges. Nodes represent variable live ranges and an edge indicates that the variables connected by that edge are simultaneously alive at some program point. A graph coloring register allocator assigns architectural registers to nodes such that two connected nodes do not receive the same color. If the graph is not colorable by N (the number of available physical registers), then some nodes are removed and the variables are spilled to memory to to make it colorable. The goal of a graph coloring allocator is to achieve an N-colorable graph with minimal spills [3, 1].

Traub et al. [18] designed linear scan allocators that greedily scan the program variables to find a register assignment. Instead of using an interference graph, they directly use the live interval, which is the collection of instructions in which the variable is live. With good heuristics for ordering variables, the compiler can achieve the same code quality as graph coloring in many cases, but significantly faster. In this study, we start with a linear scan register allocator and add bank assignment functionality to it.

Bank assignment for clustered processors. In clustered processors, functional units and register files are partitioned or replicated and then grouped into on-chip clusters [9, 12]. Clusters are connected through an inter-cluster communication network [11]. In each cluster, reads and writes are sent to the local register file (local reads/writes) or to remote register files in another cluster through the inter-cluster communication network. The register allocator attempts to minimize the number of remote register accesses.

Ellis generated code for VLIW processors with partitioned register files with a partitioning method called BUG (bottom-up greedy) intertwined with instruction scheduling [8]. Hiser et al. later extended that work by abstracting machine-dependent details into node and edge weights in a graph called the *register component graph* (RCG) [10]. The unconnected nodes (virtual registers) are good candidates to be assigned to separate banks. The algorithm first creates *an ideal instruction schedule*, assuming a single multiported register file. It then partitions the virtual registers by evaluating the benefit of assigning a given virtual register to each of the register banks, choosing the bank with the most benefit. The cost model includes the necessary copy instructions for virtual registers used in more than one partition. The algorithm runs as a pre-process before instruction scheduling and register allocation, which then use the specified banks to schedule instructions and allocate registers in partitions.

In this paper we address bank assignment for a different type of architecture in which registers, the ALUs, and the L1 cache banks all are physically partitioned and connected together via a lightweight on-chip network. The TRIPS architecture supports block atomic execution and direct dataflow communication among instructions in a block. Unlike other approaches [2, 12], register allocation must occur before scheduling in TRIPS. Because blocks have a fixed size, the compiler must insert spills before placing instructions in blocks and then schedule [6]. Cluster architectures have a fixed inter-cluster communication delay, whereas in TRIPS, the register access delay depends on the distance between the register and the ALU of the instruction reading or writing that register.

The method we propose is most similar to Hiser et al. [10], but generalizes the register component graph to consider the arrival time of the virtual registers to each instruction. The algorithm also considers physical layout of the processor grid when choosing the best bank. Whereas Hiser et al. perform bank allocation as a pre-process before register allocation, our algorithm combines bank and register assignment. Each bank allocation decision thus uses information from prior allocation decisions, including spilled registers.

3 Background

Spatially partitioned uniprocessor architectures allow higher frequency operation at lower power, while exposing greater concurrency and data bandwidth. For example, the Raw processor integrates a low-latency on-chip network into its processor pipelines in a single-chip multiprocessor [17]. It provides programmable network routers for static routing, in which the programmer treats the Raw tiles as elements of a distributed serial processor. The most important characteristic of a spatially distributed architecture is that the topology of instruction placement is exposed in the ISA and performance is greatly dependent on both the exact placement of instructions and the time spent routing data between instructions.

We investigate register banking on the TRIPS processor, which is a spatially partitioned processor that uses an EDGE ISA. In an EDGE ISA, the compiler groups instructions into large, fixed-size blocks similar to hyperblocks [13]. The

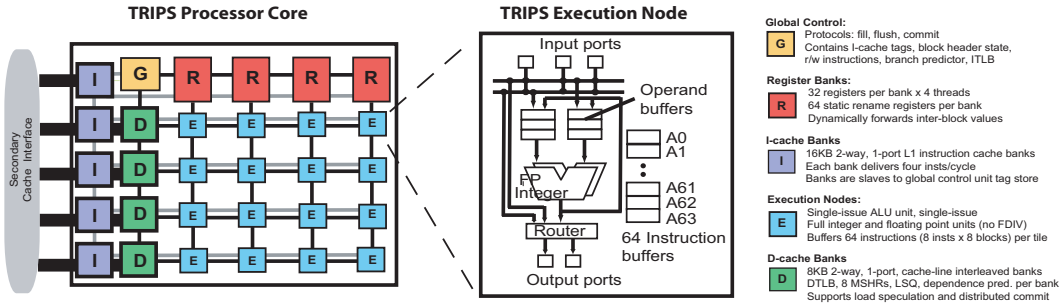

Fig. 1. A TRIPS Prototype Core

ISA employs predication to form large blocks. Within each TRIPS block, the compiler encodes the instructions in dataflow form. Each instruction specifies where to send its result. At runtime, an individual instruction executes when it receives all of its operands, and each TRIPS block is executed atomically.

3.1 Overview of TRIPS

Figure 1 shows a diagram of a TRIPS processor core composed of a 2-D grid of 16 execution tiles (ETs), 4 distributed register tiles (RTs), 4 distributed L1 data cache tiles (DTs), 5 instruction cache tiles and a global control tile. Each ET has an integer unit, a floating-point unit, and reservation stations for instructions. Each RT includes 32 registers, resulting in a total register file capacity of 128 registers. The TRIPS tiles communicate using a lightweight operand network that dynamically routes operands and load/store traffic through intermediate tiles in Y-X order.

A TRIPS program consists of a series of instruction blocks that are individually mapped onto the array of execution tiles and executed atomically [15]. The compiler statically specifies where instructions execute, i.e., on which ET. The hardware determines when they execute by dynamically issuing instructions after their operands become available. The architecture reads inputs from registers and delivers them into the ET array. Operands produced and consumed within the array are delivered directly to the target instruction. Direct instruction communication requires no register file accesses. Operand arrival triggers instruction execution, thus implementing a dataflow execution model. A TRIPS block may hold up to 128 computation instructions with up to 8 mapped to any given ET.

The TRIPS ISA imposes several constraints on the blocks generated by the compiler:

- The maximum block size is 128 instructions.
- The number of register reads and writes in a TRIPS block is 32 reads (eight reads per register tile) and 32 writes (eight writes per register tile).
- The total number of executed load and store instructions in a TRIPS block must not exceed 32.

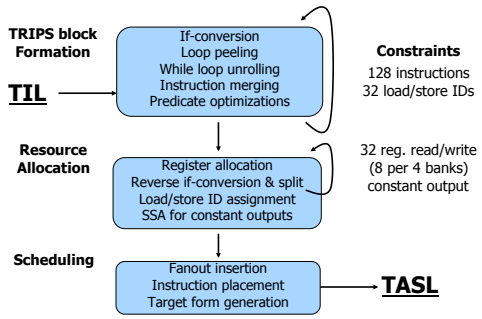

Fig. 2. TRIPS compiler overview

3.2 Compiling for TRIPS

To explain the interaction between the TRIPS register allocator and instruction scheduler, we describe different phases of the TRIPS compiler backend [15].

As shown in Figure 2, the first phase is block formation, in which the compiler combines basic blocks into a set of TRIPS blocks integrating if-conversion, predication, unrolling, tail duplication, and head duplication as necessary to form optimized blocks [13]. The compiler also performs scalar optimizations that merge redundant instructions and eliminate unnecessary predicates as an integrated step of block formation. During block formation, the compiler assumes an infinite virtual register set and uses a RISC-like intermediate form called TIL (TRIPS Intermediate Language).

After block formation, the compiler performs register allocation of virtual registers (variables) to physical registers. The variables defined and used inside a single block are not register allocated because EDGE instructions directly encode their consumer instructions. Therefore, the register allocator allocates only variables that are live-in or live-out across blocks. The allocator enforces the constraints on the TRIPS blocks regarding the number of reads and writes from each register tile. Allocation must occur prior to instruction scheduling because a spill could cause a block to violate the block size limit. In this case, the compiler performs reverse if-conversion, splits the block, and performs allocation again, until no spills violate the block constraints. We explain the register allocation algorithms in detail in the following sections.

The last phase in the TRIPS compiler backend is instruction scheduling, which outputs TRIPS Assembly Language (TASL). TASL fully specifies blocks and within blocks it encodes dataflow target form: each instruction has an identifier and each instruction may specify one or more instruction identifiers to which its result should be sent. The architecture maps instruction identifiers to execution tiles [6]. TASL is portable because the hardware can map instruction identifiers at run time based on various hardware topologies. The scheduler uses a cost function to choose an execution tile on which to place each instruction. This function considers features such as the communication among the depen-

dent instructions, network delay, and network contention. It also considers the location of the register bank (RT) of each register read or written by that instruction. The scheduler needs to know the register bank to which each variable is allocated to produce an efficient schedule.

3.3 Base Linear Scan Register Allocator

This allocator simply extends a linear scan algorithm. It performs a liveness analysis to compute the live range information at a block granularity. It sorts variables for allocation using a priority function similar to Chow and Hennessey's priority function [5]:

$$Priority_{DEF}(vr) = \sum_{i=LR(vr)} (Di * ST\_COST + Ui * LD\_COST) \qquad (1)$$

where binary values Di and Ui indicate whether the variable is defined or used in block B_i. ST_COST and LD_COST are the delays associated with store and load instructions, respectively. $LR(vr)$ returns a list of blocks in which variable vr is live. For each variable, the allocator considers all available physical registers, regardless of their register bank. For each register, the allocator tests for:

- **Live range conflicts:** The register live range must not conflict with the variable live range (i.e., the register has not already been assigned to another variable with an overlapping live range).
- **Block read/write conflicts:** The assignment must not violate the limit on the number of register reads or writes in the block (32 reads and 32 writes). Also, the assignment must not violate the limit on the number of bank accesses (8 reads and writes per bank) for all blocks that read or write the variable.

The allocator assigns the virtual register to a candidate register that meets these criteria and updates the live range of the physical register to encompass the live range of that variable. We configure this algorithm to perform bank-oblivious and round-robin assignments. Bank oblivious uses all the registers in the first bank, then the second, and so on. Round robin cycles through the banks as it assigns physical registers to variables in priority order.

If no physical register satisfies both the live range and bank conflict tests, the register allocator inserts spill loads and stores inside each block that uses or defines that virtual register. After a spill, register allocation is repeated to account for new live ranges generated by the spill code. An indirect effect of spilling on TRIPS that does not exist in conventional processors is that added loads and stores increase block sizes. If a block size exceeds the maximum (128 instructions for TRIPS), the block becomes invalid. The compiler forms valid blocks by splitting each invalid block into two blocks using reverse if-conversion [19], which creates new live ranges, and then performs register allocation again. To reduce the probability of splitting blocks, the allocator first identifies blocks that would overflow as a consequence of spilling and adds them to a list called $SIBLOCKS$

(spilling invalidate blocks). Using this list, the allocator increases the priority associated with registers used in those blocks:

$$Priority_{EDGE}(vr) = Priority_{DEF}(vr) + \alpha \sum_{i=LR(vr)\cap SIBLOCKS} \frac{SizeB_i}{128 - SizeB_i}. \quad (2)$$

where α is a fixed value greater than all possible values of $Priority_{EDGE}(vr)$, LR represents the live range of a virtual register, and $SizeB_i$ is the size of block B_i. Based on this function, any virtual register live in one of the blocks in $SIBLOCKS$ has higher priority than all virtual registers without this property.

4 Bank Assignment Algorithm for Spatially Partitioned Processors

This section explains the bank allocation algorithm for spatially partitioned processors. This algorithm, however, is not specific to the TRIPS processor and can be applied to other spatially partitioned processors. Therefore, this section explains the general algorithm, and the next section describes how we customize the algorithm for the TRIPS processor.

In spatially partitioned processors a lightweight network connects the register banks, ALUs, and data cache banks, which form a distributed 2-D substrate. A virtual register (variable) can be allocated to any of the register banks on the substrate, but the delay of accessing each register bank from an ALU on the substrate depends on the distance between the register and the ALU, as well as the contention in the network. For example, consider the sample substrate shown in Figure 3 with three ALUs ($A_{0...2}$) and three register banks ($B_{0...2}$) connected by the single delay-per-hop network shown with black lines. This figure shows two bank assignments for the variables $v_{0...3}$ where variables v_1 and v_2 are inputs to instruction i_0 in A_1 and variables v_0 and v_3 are inputs to instruction i_1 in A_2. The thick grey lines show the data transfers from register banks to the destination ALUs and the number beside each arrow indicates the arrival time of the corresponding register to that ALU. In the bank assignment on the left, the distances between variables v_0, v_1 and their destination instructions are three and two hops, respectively. However, the arrival of v_0 or v_1 will be delayed by at least one cycle because both v_0 and v_1 use the same path, i.e., the path that connects register bank B_0 to A_0. Since the network only sends one value in each cycle, one of them will be delayed. Similarly, there is a network contention between variables v_0 and v_3 in the bank assignment on the right. This bank assignment, however, places dependent variables v_0 and v_3 in the same bank, resulting in fewer network hops, lower network congestion, and better overall timing for both instructions: a minimum network delay of 2 rather than 3.

The bank assignment algorithm first creates a graph called a register dependence graph (RDG). It then chooses an order in which to attempt to allocate virtual registers to architectural registers. For every virtual register in that ordered list, it uses a bank score evaluation function to calculate the benefit of

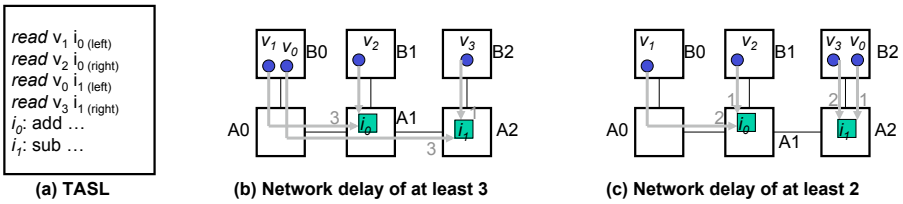

Fig. 3. Network delays resulting from different register bank assignments for a sample substrate.

placing the virtual register in each bank, and chooses the bank with the maximum score. The score for each bank is based on the banks of already allocated virtual registers and the weights of the edges of the RDG connecting that virtual register to other virtual registers. The register allocator next allocates the virtual register to one of the physical registers in that bank and the process continues.

4.1 Register Dependence Graph

The algorithm first builds a *register dependence graph* with nodes representing virtual registers and edges indicating dependences between virtual registers.

The weight on each edge between two virtual registers indicates the affinity between those two virtual registers. Lower values on an edge indicate that placing the virtual registers close together will improve the overall delay of the critical path. To create the RDG, the algorithm processes the blocks in the program one at a time. For each block, it estimates the execution time of instructions using an ideal schedule on the acyclic data flow graph (DFG) of that block. This ideal schedule assumes that all of the block's input registers arrive at the same time, and that there are no delays due to network contention. The algorithm traverses the DFG in a breadth-first order. For each instruction, it estimates the time its output virtual register is ready by adding the fixed execution delay associated with that instruction to the time when all its inputs are ready. Using this ideal schedule, the algorithm optimistically estimates when data from an input register will be available to its consumer instructions in the block DFG.

In a second pass, the algorithm traverses the DFG, keeping track of the variables that are ancestors of each instruction in the critical path. When an instruction has two different variables as ancestors, the algorithm places an edge between those variables with a weight equal to the difference between their estimated arrival times at that instruction. If an edge already exists between two virtual registers, the algorithm keeps track of the minimum weight for that edge. An edge with a low weight indicates that the two virtual registers should be placed close together.

Figure 4 provides sample intermediate code for a block, the block's DFG with the ideal estimated times, and the corresponding RDG. Variables a, b and c are the inputs to the block and must be kept in registers (other variables

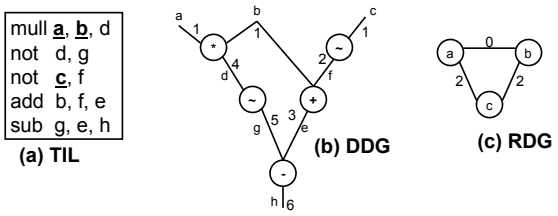

Fig. 4. (a) TIL block example, (b) Dataflow Dependence Graph (DDG) with ideal time estimates, and (c) Register Dependence Graph (RDG)

are temporary values within the block and are handled by the hardware). For this example, we assume that the execution time of a *mult* instruction is three cycles and the execution time of all other instructions is one cycle. The critical path of the block is the chain including the *mult, not,* and *sub* instructions. The chain of instructions coming from a and b intersect at the *mult* instruction at an estimated time of 1 cycle for both a and b. Because the *mult* instruction is on the critical path, we set the value on the link between a and b in the RDG to zero, meaning that the two virtual registers should be as close together as possible.

The chain of instructions originating at b intersects with the chain of instructions originating at c at the *add* instruction, and also at the sub instruction. The add instruction is not on the critical path, so this instruction is ignored. The sub instruction is on the critical path, however, so the weight of the edge between b and c is set to the difference between the arrival time of the data from each register at the *sub* instruction, $(5 - 3)$. Since we must perform scheduling after register allocation, the allocator assumes an ideal schedule to estimate latencies between instructions.

Each node in the RDG contains the following information for the corresponding virtual register:

- **Loop nesting depth:** If the virtual register is used or defined in more than one loop we select its maximum loop nesting depth.
- **Total number of instructions affected by the virtual register:** The number of instructions in the DDG which depend directly or transitively on this virtual register. For example, in Figure 4 the virtual registers a and b affect three and four instructions, respectively.

4.2 Bank Assignment

After building the RDG, the algorithm begins a combined bank assignment and register allocation phase. For a given virtual register, it first chooses the best register bank according to a heuristic function, and then tests if a physical register in that bank is available. If so, it assigns the virtual register to the physical register. Otherwise, the allocator tries to find an alternative register in another bank. Figure 5 shows the bank assignment algorithm. *PriorityOrder* determines

```
for each vr in PriorityOrder
    bestBank = 0
    bestScore = 0
    for each register bank b
        bankScore = CalculateBankScore(vr, b)
        if (bankScore > bestScore)
            bestScore = bankScore
            bestBank = b
        elsif (bankScore == bestScore)
            bestBank = TieBreak(vr, bestBank, b)
    reg = ChoosePhysicalRegisterFromBank(bestBank, vr)
    if (reg found)
        Replace vr with r in the code and update data for vr and bestBank
    else
        reg = ChoosePhysicalRegisterFromOtherBanks(bestBank, vr)
        if (reg found)
            Replace vr with r in the code and update data for vr and bestBank
        else
            Spill vr
```

Fig. 5. Bank Assignment Algorithm

the order in which the virtual registers are allocated. In classic register allocation studies, the priority for each virtual register is computed based on spill code overhead produced if that virtual register is spilled [4, 5]. Because bank assignment is done in conjunction with register allocation, however, the priority order must also take into account the dependencies between virtual registers and their criticality. We define a simple priority function as follows:

$$Priority_{spatial}(vr) = 10^{LoopNestingDepth} + NumOfEdges(vr, RDG). \quad (3)$$

This function prioritizes virtual registers that have more dependencies with other virtual registers and affect more instructions in the DDG.

The bank allocation algorithm uses the *CalculateBankScore* cost function.Figure 6 shows a basic implementation of this function that uses the following components:

- *Dependence score*: A score based on the dependencies between the current virtual registers and the already allocated virtual registers. The function accumulates the weights of the RDG edges between the current virtual register and all virtual registers assigned to the current bank and its neighbor banks (referred to as *NeighborBankSet* in Figure 6).
- *Bank utilization penalty*: The number of registers already assigned to the bank. This component favors distributing virtual registers evenly across the banks to improve concurrency.

```
CalculateBankScoreBasic(vr, bank)
return CalculateDependenceScore(vr, bank) - bank.numAssignedVR

CalculateDependenceScore(vr, bank)
score = 0
for each nvr RDG neighbor of vr assigned to NeighborBankSet(bank)
    score += (RDG-MAX-WEIGHT - RDG Weight(vr, nvr))
return score
```

Fig. 6. Basic Implementation of CalculateBankScore Function

```
TRIPS_TieBreaker(vr, bank1, bank2)
if (vr.affectedCriticalLoads + vr.affectedCriticalStores > 0)
    return min(bank1, bank2)
else
    return max(bank1, bank2)
```

Fig. 7. TieBreaker Function for TRIPS

The algorithm uses a tie breaker function, *TieBreak*, to determine the bank when the scores of two banks for a given virtual register are identical. The definition of *NeighborBankSet* and *TieBreaker* functions depends on the physical layout and the characteristics of the processor grid. We explore implementations of these functions for TRIPS processor.

4.3 Customizing the Bank Score Evaluation Function for TRIPS

Because TRIPS has fewer register banks than execution tiles, we expect heavy traffic on the links connecting register banks to the execution tiles, and contention on those links affects performance. We implement a *TieBreaker* function to separate the load/store traffic, which flows from register banks to the data tiles, from the rest of the traffic, as shown in Figure 7. This function prioritizes register banks on the left (i.e., lower bank numbers) if there are critical load or store instructions dependent on the arrival time of the current virtual register. Otherwise, it prioritizes the register banks on the right to move the non-memory traffic out of the way of traffic to the data cache banks to avoid network contention.

5 Experimental Results

To evaluate performance we used the TRIPS hardware. The TRIPS chip is a custom 170 million transistor ASIC implemented in a 130nm technology. For these experiments, we ran the processor at 366MHz. The capacity of the L1 cache, L2 cache and main memory were 32 KB, 1 MB and 2GB, respectively. We

collected cycle counts from the hardware performance counters using customized libraries and a runtime environment developed by the TRIPS team [20]. We used C and Fortran programs from the EEMBC benchmark suite with the iteration count set to 1000, and from the SPEC2000 benchmark suite [7, 16].

For comparison, we adapted Hiser et al.'s algorithm (HCSB) to work with an EDGE ISA. We add a link in the RDG between two virtual registers if one is the input to a hyperblock, the other is an output from that same hyperblock, and there exists a dataflow path within the hyperblock from the input virtual register to the output virtual register. For example, consider the block in Figure 4. There will be edges between h and each of the inputs (a, b and c) in the RDG. The remaining operations from HCSB carry over without changes to an EDGE ISA.

We compare the following bank alloction algorithms on TRIPS:

- **Bank Oblivious:** The linear scan register allocation algorithm explained in Section 3.3 with no bank assignment mechanism. This algorithm uses all of the architectural registers in the current bank before using the registers in the next bank.
- **Round Robin:** The linear scan register allocation algorithm explained in Section 3.3 using round-robin bank allocation. This allocator chooses physical registers from banks in a round robin fashion.
- **HCSB:** Our implementation of Hiser et al. [10].
- **Spatial:** The bank allocation algorithm for spatially partitioned processors using the bank allocation priority function shown in Equation 3 and the basic bank score function shown in Figure 6.

Table 1 contains the number of static spill load and store instructions for each of the four allocators. Register allocation adds spill code to only 5 benchmarks out of the 39 EEMBC and SPEC benchmarks we evaluated. This low rate of spill code generation is partly because TRIPS has more registers than conventional architectures. In addition, the TRIPS compiler converts temporary values defined and used within a block to direct instruction communication that does not go through the register file, as it would on a conventional RISC processor.

For the benchmarks in Table 1, the spatial and HCSB bank allocators produce less spill code compared to the bank oblivious and round-robin allocators. The spatial and HCSB allocators prioritize variables based on the number of dependences according to the priority function shown in Equation 3. As a result, they allocate critical variables with higher numbers of dependences first. These dependences directly indicate the number of spill instructions. By prioritizing the variables with the most dependences first, if a variable later must be spilled, then it will likely have fewer dependences and thus require less spill code.

Figure 8 provides speedups using different bank assignment algorithms relative to the performance of the bank oblivious allocator. On average, the spatial bank assignment outperforms the other register allocators. The round-robin algorithm achieves a 3% performance improvement over the bank oblivious algorithm. This performance improvement results from a more balanced register distribution. HCSB improves performance over the round-robin bank allocation

Table 1. Number of static spill load and store instructions. For the remaining benchmarks, none of the allocators generate any spill instructions.

Program	Benchmark suite	Bank oblivious	Round robin	HCSB	Spatial
a2time	EEMBC	111	111	30	31
applu	SPEC	528	514	365	382
apsi	SPEC	328	220	183	183
equake	SPEC	30	30	10	10
mgrid	SPEC	44	21	8	12

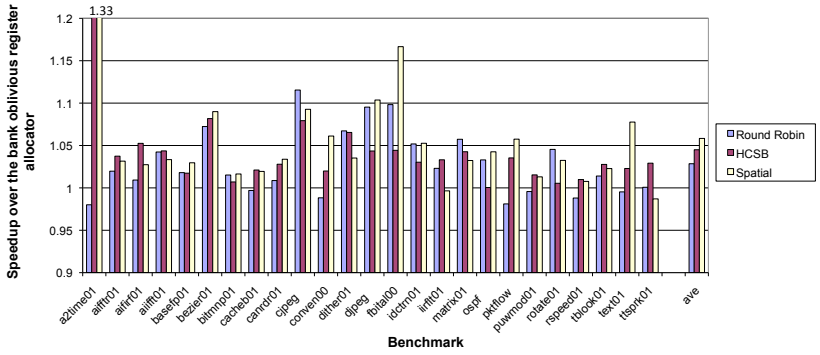

Fig. 8. The speedup achieved using different bank assignment algorithms for *EEMBC* on TRIPS compared to a bank-oblivious linear scan allocator

by placing dependent variables in nearby register banks. This algorithm, however, does not consider the arrival times of variables or the topology of the processor substrate. The spatial bank assignment algorithm places the virtual registers in the register banks according to their arrival times at the critical instructions. It also places registers used by critical load or store instructions to the register banks located close to the cache banks. On average, this bank assignment algorithm performs 6% better than the bank oblivious assignment.

For some programs, the spatial bank allocator performs significantly better than other allocators. The a2time benchmark has the largest speedup when using the spatial or HCSB allocators. Table 1 shows that a2time is the only EEMBC benchmark for which spill code is generated, and that the HCSB and spatial algorithms significantly reduce spilling for this benchmark.

In fbital, the spatial bank allocator achieves a high speedup over the bank oblivious allocator, and the round-robin allocator achieves the second best speedup. Figure 9 (a) illustrates the simplified version of the most frequently executed block of this program. In the critical path of this block (the grey lines in the figure), the computation chain starting from two virtual registers v_1 and v_2 ends by writing to variable v_2. Variables v_0 and v_1 are also inputs to a store mem-

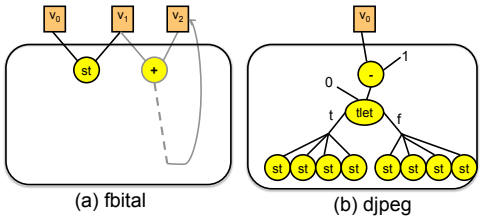

Fig. 9. The critical paths of fbital and djpeg EEMBC programs

ory operation. By separating memory and computation traffic, the spatial bank allocator places v_0, v_1 and v_2 in banks 0, 2, and 3, respectively. However, the HCSB bank allocator, which considers only register dependencies, places these dependent variables in banks 2, 1, and 1, respectively. Because of this allocation, the critical path suffers extra delays caused by the memory traffic of the store instruction. Round robin randomly places v_1 and v_2 in banks 2 and 3, achieving better results than HCSB.

In the critical path of djpeg, a predicate condition is computed using an input variable, as shown in Figure 9 (b). Several parallel memory operations are executed on both predicate paths. The round-robin bank allocator places the critical variable in bank 0, which adds some delays to the predicate computations because of the high memory traffic. The HCSB allocator places that variable in bank 3, which is too far from the memory banks and adds some delays to the memory operations. Considering memory bank locations, the spatial bank allocator places that variable in bank 2, which results in the highest speedup over the bank oblivious allocator.

For some programs, round robin and HCSB perform better than spatial. Examples of such programs are aifirf01, aiifft01 and matrix01. We suspect that in these programs, the ideal schedule model used by the spatial bank assignment algorithm to generate the register dependence graph has inaccuracies caused by runtime resource constraints such as long latency cache misses.

Figure 10 illustrates the speedups using different bank assignment allocators relative to the performance achieved using the bank oblivious allocator for the SPEC benchmarks. Register allocation does not affect the SPEC benchmarks as strongly as it affects the EEMBC benchmarks, most likely because the SPEC benchmarks are limited by other factors such as memory latency or instruction cache pressure. As a result, benchmarks such as gzip, bzip, and equake are not strongly affected. HCSB and round robin perform similarly on average, while the spatial bank allocator performs slightly better. This speedup may be the result of separating memory traffic from computation traffic.

6 Conclusions

In spatially partition processors, the delay of accessing registers depends on the location of the register files and the ALUs on the grid. Consequently, we in-

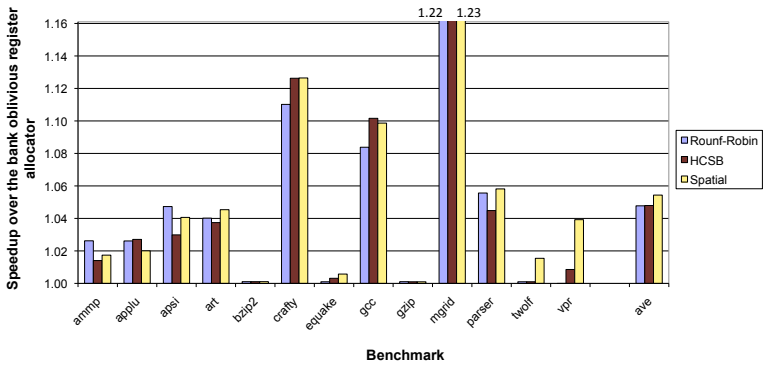

Fig. 10. The speedup achieved using different bank assignment methods for the *SPEC* benchmarks on TRIPS

troduce a register allocator that avoids spilling critical registers and estimates operand arrival times for critical instructions to choose banks for dependent registers wisely. The allocator also considers the topological characteristics of the hardware. In TRIPS, the memory tiles are located on the left side of the grid. Considering the location of register files, the spatial allocator separates memory traffic from computation traffic when assigning banks to critical registers. In addition, this allocator places dependent critical registers close together, so that critical instructions receive their operands faster. The individual effects of proximity of dependent registers and separation of memory and computation traffic is still an open question and requires further research.

7 Acknowledgments

We thank Jon Gibson and Aaron Smith who implemented the round-robin allocator. We also thank the entire TRIPS hardware and compiler teams.

This work is supported by DARPA F33615-03-C-4106, NSF EIA-0303609, Intel, IBM, and an NSF Graduate Research Fellowship. Any opinions and conclusions are the authors' and do not necessarily reflect those of the sponsors.

References

1. D. Bernstein, M. Golumbic, Y. Mansour, R. Pinter, D. Goldin, and I. Nahshon H. Krawczyk. Spill code minimization techniques for optimizing compliers. In *ACM SIGPLAN Symposium on Interpreters and Interpretive Techniques*, pages 258–263, 1989.
2. T. S. Brasier, P. H. Sweany, S. J. Beaty, and S. Carr. CRAIG: a practical framework for combining instruction scheduling and register assignment. In *Parallel Architectures and Compilation Techniques*, pages 11–18, 1995.

3. P. Briggs, K. D. Cooper, and L. Torczon. Improvements to graph coloring register allocation. In *ACM Transactions on Programming Languages and Systems, 16(3)*, pages 428–455, May 1994.
4. G. Chaitin. Register allocation and spilling via graph coloring. In *ACM SIGPLAN Symposium on Compiler Construction*, pages 98–105, 1982.
5. F. C. Chow and J. L. Hennessy. Priority-based coloring approach to register allocation. In *ACM Transactions on Programming Languages and Systems, Vol. 12*, pages 501–536, 1990.
6. K. Coons, X. Chen, S. Kushwaha, D. Burger, and K. S. McKinley. A spatial path scheduling algorithm for edge architectures. In *ACM Conference on Architecture Support for Programming Languages and Operating Systems*, pages 129–140, 2006.
7. EEMBC. Embedded microprocessor benchmark consortium, http://www.eembc.org/.
8. J. Ellis. *A Compiler for VLIW Architecture*. PhD thesis, Yale University, 1984.
9. K. L. Farkas, P. Chow, N. P. Jouppi, and Z. Vranesic. The multicluster architecture: Reducing processor cycle time through partitioning. In *ACM/IEEE Symposium on Microarchitecture*, pages 327–356, 1997.
10. J. Hiser, S. Carr, P. Sweany, and S. J. Beaty. Register assignment for software pipelining with partitioned register banks. In *International Parallel and Distributed Processing Symposium*, pages 211–217, 2000.
11. J. Janssen and H. Corporaal. Partitioned register files for TTAs. In *ACM/IEEE Symposium on Micorarchitecture*, pages 301–312, December 1995.
12. K. Kailas, K. Ebcioglu, and A. Agrawala. Cars: A new code generation framework for clustered ILP processors. In *Conference on High Performance Computer Architecture*, pages 133–143, 2001.
13. B. Maher, A. Smith, D. Burger, and K.S. McKinley. Merging head and tail duplication for convergent hyperblock formation. In *ACM/IEEE International Symposium on Microarchitecture*, pages 65–76, 2006.
14. M. Poletto and V. Sarkar. Linear scan register allocation. In *ACM Transactions on Programming Languages and Systems, Vol. 21*, pages 895–913, Spetember 1999.
15. A. Smith, J. Burrill, J. Gibson, B. Maher, N. Nethercote, B. Yoder, D. C. Burger, and K.S. McKinley. Compiling for EDGE architectures. In *International Conference on Code Generation and Optimization*, pages 185–195, 2006.
16. SPEC2000CPU. The standard performance evaluation corporation (SPEC), http://www.spec.org/.
17. M. B. Taylor and A. Agarwal. Evaluation of the raw microprocessor: An exposed-wire-delay architecture for ILP and streams. In *ACM SIGARCH International Symposium on Computer Architecture*, pages 2–13, 2004.
18. O. Traub, G. Holloway, and M. D. Smith. Quality and speed in linear-scan register allocation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 895–913, June 1998.
19. N. J. Warter. Reverse if-conversion. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 290–299, 1993.
20. B. Yoder, J. Burrill, R. McDonald, K. B. Bush, K. Coons, M. Gebhart, S. Govindan, B. Maher, R. Nagarajan, B. Robatmili, K. Sankaralingam, S. Sharif, and A. Smith. Software infrastructure and tools for the TRIPS prototype. In *Third Annual Workshop on Modeling, Benchmarking and Simulation*, 2007.