

# Register Assignment for Software Pipelining with Partitioned Register Banks \*

Jason Hiser<sup>†</sup>

Steve Carr<sup>‡</sup>

Philip Sweany<sup>‡</sup>

Steven J. Beaty<sup>§</sup>

## Abstract

*Many techniques for increasing the amount of instruction-level parallelism (ILP) put increased pressure on the registers inside a CPU. These techniques allow for more operations to occur simultaneously at the cost of requiring more registers to hold the operands and results of those operations, and importantly, more ports on the register banks to allow for concurrent access to the data. One approach of ameliorating the number of ports on a register bank (the cost of ports in gates varies as  $N^2$  where  $N$  is the number of ports, and adding ports increases access time) is to have multiple register banks with fewer ports, each attached to a subset of the available functional units. This reduces the number of ports needed on a per-bank basis, but can slow operations if a necessary value is not in an attached register bank as copy operations must be inserted. Therefore, there is a circular dependence between assigning operations to functional units and assigning values to register banks. We describe an approach that produces good code by separating partitioning from scheduling and register assignment. Our method is independent of both the scheduling technique and register assignment method used.*

## 1. Introduction

With aggressive instruction scheduling techniques and significant increases in instruction-level parallelism (ILP), modern computer architectures have seen impressive performance increases. Unfortunately, these factors put large demands on a machine's register resources by requiring many data accesses per cycle. The number of ports required for such register banks severely hampers access time

[3, 7] and cost. Partitioned register banks are one mechanism for providing high degrees of ILP while maintaining a high clock rate and a reasonable cost. Texas Instruments, for example, already produces several DSP chips that have partitioned register banks to support high ILP [14]. Unfortunately, partitioned register banks may inhibit ILP as some mechanism is required to allow functional units access to "non-local" values, i.e.: values contained in a different partition. One approach to provide non-local register values is to add extra instructions to move the data to the register bank of the target functional unit. Another approach is to provide a communication network to allow access to non-local values. In either case, a compiler must deal not only with achieving maximal parallelism via aggressive scheduling, but also with data placement among a set of register banks.

A compiler for an ILP architecture with partitioned register banks must decide for each operation, not only where the operation fits in an instruction schedule, but also in which register partition(s) the operands of that operation will reside. This, in turn, will determine which functional unit(s) can perform the operation. Obtaining an efficient assignment of operations to functional units is not an easy task as two opposing goals must be balanced. One, achieving near-peak performance requires spreading the computation over the functional units equally, thereby maximizing their utilization. Two, the compiler must minimize costs resulting from copy operations introduced by distributing the computation.

Some of the work in the areas of code generation for ILP architectures with partitioned register banks has dealt with global scheduling techniques, but most consider only local scheduling. It is well accepted that extensive loop optimization is necessary to take advantage of significant program ILP. Perhaps the most popular of loop optimization techniques currently in use in the ILP compiler community is software pipelining [9, 12, 1]. Therefore, while our register partitioning method is applicable to entire programs, we will concentrate on software pipelined loops for the work in this paper. There are at least two reasons to restrict ourselves to software pipelined loops. One is that because most of a program's execution time is typically spent in loops, loop optimization is particularly important for good code

\* This research was supported by NSF grant CCR-9870871 and a grant from Texas Instruments.

<sup>†</sup> Department of Computer Science, University of Virginia, Charlottesville VA 22903, [jdh8d@virginia.edu](mailto:jdh8d@virginia.edu).

<sup>‡</sup> Department of Computer Science, Michigan Technological University, Houghton MI 49931-1295, [{carr,sweany}@mtu.edu">carr,sweany}@mtu.edu](mailto).

<sup>§</sup> Metropolitan State College of Denver, Department of Mathematical and Computer Sciences, Campus Box 38, P. O. Box 173362, Denver, CO 80217-3362, [beatys@mscd.edu](mailto:beatys@mscd.edu)

generation. Two, and perhaps more importantly from the perspective of this paper, is that software pipelining leads both to the greatest extraction of parallelism within a program and also places significant requirements on the register resources. Considering the combination of maximal parallelism and large register requirements, we expect partitioning of software pipelined loops to be a particularly difficult problem, as well as an important one to solve.

Register banks can be partitioned such that one register bank is associated with each functional unit or by associating a cluster of functional units with each register bank. In general, we would expect that cluster-partitioned register banks would allow for “better” allocation of registers to partitions (fewer copies would be needed and, thus, less degradation compared to an ideal ILP model.) This comes at the expense of adding to the difficulty of assigning registers within each partition as additional pressure is put on them by increased parallelism.

Previous approaches to this problem have relied on building a *directed acyclic graph* (DAG) that captures the precedence relationship among the operations in a program segment. Various algorithms have been proposed on how to partition the nodes of this “operation” DAG, to generate an efficient assignment of functional units. In contrast, our approach allocates registers to partitioned register banks based on the partitioning of an undirected graph that interconnects those program data values that appear in the same operation. This graph allows us to support retargetability by abstracting machine-dependent details into node and edge weights. We call this technique *register component graph* (RCG) partitioning, because the nodes of the graph represent virtual registers appearing in the program’s intermediate code.

The experimental work included here studies software pipelined loops. A good summary of software pipelining can be found elsewhere [1]. In this paper, we begin in Section 2 with a look at others’ attempts to generate code for ILP architectures with partitioned register banks. Section 3 describes our general code-generation framework for partitioned register banks while Section 4 outlines the greedy heuristic we use in register partitioning. Finally, Section 5 describes our experimental evaluation of our partitioning scheme and Section 6 summarizes and suggests areas for future work.

## 2. Other Partitioning Work

Ellis [6] describes an early solution to the problem of generating code for partitioned register banks in his dissertation. His method called BUG (bottom-up greedy), is applied to a scheduling context at a time (*e.g.*, a trace) and is intimately intertwined with instruction scheduling, utilizing machine-dependent details within the partitioning algorithm. Our method abstracts away machine-dependent de-

tails from partitioning with edge and node weights, a feature extremely important in the context of a retargetable compiler.

Capitiano et al. [3] present a code-generation technique for limited-connectivity VLIWs. Their results (for two of the seven software-pipelined loops they tested) for three functional units, each with a dedicated register bank, show degradation in performance of 57% and 69% over code obtained for a single register bank.

Janssen and Corporaal propose an architecture called a Transport Triggered Architecture (TTA) that has an interconnection network between functional units and register banks so that each functional unit can access each register bank [8]. They report results that show significantly less degradation than the partitioning scheme of Capitiano et al., however their interconnection network represents a different architectural paradigm making comparisons less meaningful. Indeed it is surmised that their interconnection network would likely degrade processor cycle time significantly, making this architectural paradigm infeasible for hardware supporting the high levels of ILP where maintaining a single register bank is impractical. Additionally, chip space is limited and allocating space to an interconnection network may be neither feasible nor cost effective.

Ozer, et al., present an algorithm, called *unified assign and schedule* (UAS), for performing partitioning and scheduling in the same pass [11]. They state that UAS is an improvement over BUG since UAS can perform schedule-time resource checking while partitioning allowing UAS to manage the partitioning with the knowledge of the bus utilization for copies between partitions. Ozer’s study of entire programs showed, for their best heuristic, an average degradation of roughly 19% on an 8-wide machine grouped as two clusters of 4 functional units and 2 busses.

Nystrom and Eichenberger [10] present an algorithm that first performs partitioning with heuristics that consider later modulo scheduling. Specifically, they try to prevent inserting copies that will lengthen the recurrence constraint of modulo scheduling. If copies are inserted off of critical recurrences in recurrence-constrained loops, the initiation interval for these loops may not be increased if enough copy resources are available. Our greedy partitioning method is not limited to just software pipelined loops but is easily applicable to entire programs, since we could easily use both non-loop and loop code to build our register component graph. Nystrom and Eichenberger restrict themselves to loops and their method includes facets that makes it difficult to apply to larger entities.

### 3. Register Assignment with Partitioned Register Banks

A good deal of work has investigated how registers should be assigned when a machine has a single “bank” of equivalent registers [4, 5, 2]. However, on architectures with high degrees of ILP, it is often inconvenient or impossible to have a single register bank associated with all functional units as it would require too many read/write ports to be practical [3]. Consider an architecture with a rather modest ILP level of six. This means that we wish to initiate up to six operations each clock cycle. As each operation could require up to three registers (two sources and one destination) such an architecture would require simultaneous access of up to 18 different registers (12 for reading and 6 for writing, although reading and writing may be separated in the execution cycle) from the same register bank. An alternative to the single register bank for an architecture is to have a distinct set of registers associated with each functional unit (or cluster of functional units). Examples of such architectures include the Multiflow Trace and several chips manufactured by Texas Instruments for digital signal processing [14]. Operations performed in any functional unit require registers with the proper associated register bank, and copying a value from one register bank to another is expensive. The problem for the compiler, then, is to allocate registers to banks to reduce the number of copies, while retaining a high degree of parallelism.

Our approach to this problem is as follows.

1. Build intermediate code with symbolic registers, assuming a single infinite register bank.
2. Build data dependence graphs (DDGs) and perform software pipelining still assuming an infinite register bank.
3. Partition the registers to register banks (and thus preferred functional unit(s)) by the “Component” method outlined below.
4. Re-build DDGs and perform instruction scheduling attempting to assign operations to the “proper” (cheapest) functional unit based upon the location of the registers.
5. With functional units specified and registers allocated to banks, perform “standard” Chaitin/Briggs graph coloring register assignment for each register bank.

#### 3.1. Partitioning Registers by Components

Our method builds a graph, called the *register component graph*, whose nodes represent register operands (symbolic registers) and whose arcs indicate that two registers

“appear” in the same (atomic) operation. Arcs are added from the destination register to each source register. We build the register component graph with a single pass over either the intermediate code representation of the function being compiled, or alternatively, with a single pass over scheduled instructions. We find it useful to build the graph from what we call an “ideal” instruction schedule which by our definition, uses all the characteristics of the actual architecture, except that it assumes that all registers are in a single multi-ported register bank. Once the register component graph is built, values that are not connected in the graph are good candidates to be assigned to separate register banks. Once the graph is built, we find each connected component of the graph as each represents registers that can be allocated to a single partition. In general, we need to split components to fit the number of register partitions available on a particular architecture, computing a cut-set of the graph that has a low copying cost and high degree of parallelism among components.

A major advantage of the register component graph is that it abstracts away machine-dependent details into costs associated with the nodes and edges of the graph. This is extremely important in the context of a retargetable compiler that needs to handle a wide variety of machine idiosyncrasies, such as when  $A = B \text{ op } C$  where each of  $A$ ,  $B$  and  $C$  must be in separate register banks. This situation could be handled abstractly by weighting the edges connecting these values with negative values of “infinite” magnitude, thus ensuring that the registers are assigned to different banks. An even more idiosyncratic example, but one that exists on some architectures, would require that  $A$ ,  $B$ , and  $C$  not only reside in three different register banks, but specifically in banks  $X$ ,  $Y$ , and  $Z$ , and furthermore that  $A$  use the same register number in  $X$  that  $B$  does in  $Y$  and that  $C$  does in  $Z$ . Needless to say, this complicates matters. By pre-coloring [4] both the register bank choice and the register number choice within each bank, however, it can be accommodated within our register component graph framework.

#### 3.2. A Partitioning Example

To demonstrate the register component graph method of partitioning, we start with the following software pipelined inner loop from matrix multiply, assuming a single cycle latency for add, and a two-cycle pipeline for multiply. In addition we’ll assume that all loads are cache hits and also use a two-cycle pipeline. An ideal schedule for this inner loop is shown in Figure 1. The corresponding register component graph appears in Figure 2. The loop kernel requires 4 cycles to complete.<sup>1</sup>

<sup>1</sup>we assume the availability of a load command that allows for either pre-increment or post-increment of the address operand.







